

Level 0 Training

Session 2

Arrays and Strings

Store collections. Work with text.

A step-by-step reference to read, trace,
practise, and return to after the session.

How to use this reference

This material is written for studying **after the training session**. It explains the ideas behind the code, not only the syntax. Read a little, work through a small example, and then write a program of your own.

A study routine that makes the ideas stick

1. **Read and explain.** Say what the new idea does in one sentence. Explain each unfamiliar word before continuing.
2. **Trace before running.** Draw the array or string. Label its indices and record the variables that change.
3. **Type and experiment.** Run a complete example, then change one input. Predict the new answer first.
4. **Practise independently.** Attempt a problem, then use its hint. Read the full solution only after a genuine attempt, and rewrite it later without looking.

How examples are labelled

A **complete program** contains its headers and `main`; it can run by itself. A short fragment belongs **inside main** unless another location is stated. **Wrong example** means deliberately incorrect teaching code: do not copy it as a template.

Pause and predict questions have an explained answer nearby. Cover it first. **Optional extension** topics are a second-reading route; the core practice does not require them.

The language and source files

The book uses **C++23** with standard headers and readable code. Every complete program is included as a separate `.cpp` file in the LaTeX source package. The printed listings are loaded directly from those files. Each program has its own `main`; compile one file at a time.

Study one stage at a time

Session 2 assumes you can read input, use variables, write `if` statements, and trace `for` and `while` loops. Revisit the corresponding Session 1 topic whenever one of those steps feels unclear.

Use the quick-reference pages for revision *after* reading the explanations. Getting a small example wrong is useful: it tells you exactly what to investigate next.

The Level 0 learning path

Session	Topics
1	Intro to problem solving: data types, conditions, and loops.
2	Arrays and strings.
3	Functions and complexity analysis.
4	STL, part 1: vector, pair, deque, queue, priority queue, and stack.
5	STL, part 2: set, multiset, ordered set, ordered map, and unordered map.
6	Static range queries: prefix sum, partial sum, and frequency array.

What is included in this edition

This booklet contains **Session 2: Arrays and Strings**. Its section numbers begin with 2 so they match the collected book. The updated collected edition contains both Sessions 1 and 2.

What Session 2 will help you do

Store and revisit a list of numbers. Separate a value from its position. Search, update, compare, and reverse a sequence. Work with rows and columns. Read a word or a full line correctly, then process its characters to count, filter, compare, or build text.

What can wait

Custom functions and formal complexity analysis belong to Session 3. Vector and the other listed containers belong to later sessions. Frequency arrays and prefix sums are left for Session 6. This session does not depend on those topics.

Conventions for this chapter

Input follows the stated constraints. Each example states whether an empty sequence is allowed. Character exercises use ordinary ASCII letters, digits, and punctuation unless stated otherwise. Positions in statements may be one-based, but stored elements use zero-based indices; read the output requirement carefully.

Contents

How to use this reference	i
The Level 0 learning path	ii
2 Arrays and Strings	1
2.1 Why do we need arrays?	2
2.2 Declaring and initializing an array	3
2.3 Indices, capacity, and safe boundaries	5
2.4 Reading and printing an array	7
2.5 Traversing: sums, counts, minimum, and maximum	9
2.6 Searching: a value is not its position	12
2.7 Updating, copying, and comparing arrays	14
2.8 Neighbours, reversal, and ordered sequences	16
2.9 Two-dimensional arrays: rows and columns	19
2.10 Square matrices: diagonals and transposition	22
2.11 Characters: letters, digits, and single quotes	24
2.12 Getting started with std::string	26
2.13 Reading a word, a line, or a line after a number	29
2.14 Traversing strings: counting and filtering	32
2.15 Joining and comparing strings	35
2.16 Substrings and finding text	37
2.17 Palindromes and reversing text	39
2.18 Counting words: remember whether you are inside one	41
2.19 Multiple test cases: reset state, not the wrong input	43
2.20 Worked problems: combine arrays, strings, and conditions	45
2.21 Optional extension: sorting and reversal helpers	48
2.22 Optional extension: shifting, rotating, inserting, and deleting	50
2.23 Optional extension: subarrays, substrings, and subsequences	53
2.24 Optional extension: other useful array and string tools	55
2.25 Debugging arrays and strings	59
2.26 Practice problems	61
2.27 Hints before solutions	66
2.28 Explained practice solutions	68
2.29 Quick reference, glossary, and revision checklist	89
2.30 Technical notes and references	93

SESSION 2

Arrays and Strings

Remember many values. Work with a sequence of characters.

In Session 1, a variable stored one value and a loop repeated an action. Now we put those ideas together: an **array** stores a collection of values, and a **string** stores text. Loops let us visit their elements, one at a time.

You will still use the same building blocks: input, variables, arithmetic, conditions, and loops. The new question is: *which element am I using right now?*

The three ideas to keep in mind

Idea	Meaning	Example
Value	The information stored in a cell.	The score 90.
Index	A zero-based label used to access a cell.	Index 0 is the first cell.
Length	How many elements belong to the sequence.	A list of 5 scores has length 5.

Your route through the chapter

Stage A: numbers. Learn array declarations, valid indices, input/output, scans, searching, updates, and reversal. Then use two indices for a table.

Stage B: text. Distinguish a character from a string, choose the right input method, and practise counting, filtering, comparing, and checking palindromes.

Stage C: independent work. Study the worked problems, try the practice, and use the hints before the full solutions. The topics marked **Optional extension** can wait until a second reading.

Before you begin

Be able to trace a loop such as `for (int i = 0; i < n; ++i)` and explain `if`, `break`, `&&`, and `||`. You do not need custom functions, vector, frequency arrays, or formal complexity analysis for this chapter.

2.1 Why do we need arrays?

One variable cannot remember a whole history

Suppose you read four scores into the same variable:

```
int score = 0;
for (int i = 0; i < 4; ++i) {
    std::cin >> score;
}
```

After input 12 7 20 9, `score` contains 9. Each new input replaced the previous value. This is fine when you only need a running total, but not when you later need to print all four scores in reverse order.

You could declare `score1`, `score2`, and so on. That becomes awkward when there are hundreds of scores. More importantly, a loop cannot create a variable name by attaching its counter to the word `score`.

An array gives related values one shared name

An **array** stores a fixed number of elements of the same type. Each element has an **index**: a number telling us which cell to access. In C++, the first index is 0. Array elements occupy consecutive storage locations.^[2-1]

```
int scores[4] = {12, 7, 20, 9};
```

Index	0	1	2	3
Stored value	12	7	20	9

`scores[2]` means the element at index 2, whose value is 20. It does **not** mean “find the value 2.”

When storage is useful, and when it is not

To print the values backwards, compare distant elements, or revisit scores after computing an average, keeping the list is useful. To calculate only a total, you can still read and add one value at a time as in Session 1. An array is a tool, not a requirement for every problem with several inputs.

Pause and predict – 1

For `int a[4] = {9, 3, 8, 2};`, what are `a[0]`, `a[2]`, and the last valid index?

Check your prediction. They are 9, 8, and 3. There are four elements, but their indices are 0, 1, 2, and 3.

2.2 Declaring and initializing an array

Read the declaration from left to right

```
int a[5] = {4, 7, 1, 9, 2};
```

`int` is the element type. `a` is the shared name. `[5]` creates five cells. The values inside braces initialize those cells in order. End the declaration with a semicolon.

The number in the declaration is a **count**, not an index to access. In a later expression such as `a[3]`, the number is an **index**. The same square-bracket symbols have different jobs in these two places.

Useful initialization forms

```
int a[5] = {4, 7, 1, 9, 2}; // All five values are given.
int b[5]{};                // All five values start at zero.
int c[5] = {6, 2};         // 6, 2, 0, 0, 0.
int d[] = {8, 3, 5};       // Size is inferred: three cells.
```

For these integer arrays, a brace initializer with too few values initializes the remaining cells to zero. Too many values for the declared size is an error.^[2-2]

A common trap: braces do not repeat one value

`int a[5] = {-1};` produces -1, 0, 0, 0, 0, not five copies of -1. To fill every active element with a nonzero value, assign it in a loop.

```
int a[5]{};
for (int i = 0; i < 5; ++i) {
    a[i] = -1;
}
```

Do not read an uninitialized cell

An ordinary local declaration such as `int a[5];` does not initialize its integer elements. Reading an element before assigning a value is not a way to obtain a random number; it can cause undefined behavior in C++²³.^[2-3]

Prefer `int a[5]{};` when learning. Even then, remember that an initial zero is not automatically a real input value. Only the cells belonging to the current input should be processed.

Other element types use the same idea

```
long long totals[4]{};
double measurements[3] = {1.5, 2.0, 3.25};
char letters[3] = {'H', 'N', 'U'};
bool chosen[4]{}; // All elements start as false.
```

Each individual array has one element type. Choose it using the allowed values, just as you would choose the type of a single variable.

Access, update, and print

Complete program E01 – access and update five scores

```
#include <iostream>

int main() {
    int scores[5] = {12, 7, 20, 9, 15};
    std::cout << scores[0] << ' ' << scores[4] << '\n';
    scores[1] = scores[1] + 3;
    for (int i = 0; i < 5; ++i) {
        if (i > 0) { std::cout << ' '; }
        std::cout << scores[i];
    }
    std::cout << '\n';
    return 0;
}
```

```
12 15
12 10 20 9 15
```

The first output uses the first and last elements. The assignment changes only index 1 from 7 to 10. In the final loop, `i` changes, so `scores[i]` refers to a different cell on each repetition.

Pause and predict – 2

What values does `int a[4] = {5};` contain? How would you make all four elements equal to 5?

Check your prediction. It contains 5, 0, 0, 0. Assign `a[i] = 5;` for each index from 0 through 3, or write all four initial values explicitly.

2.3 Indices, capacity, and safe boundaries

Allocated capacity is not the current input length

Imagine an array with room for 1000 values, but a problem instance gives only 4 values. The **capacity** is 1000; the **active length** is 4. We store that active length separately in **n**.

```
const int CAPACITY = 1000;
int a[CAPACITY]{};
int n = 0;
std::cin >> n; // This example requires 0 <= n <= 1000.
```

CAPACITY is a named constant initialized by a compile-time value. The array's allocated size is fixed for its lifetime. Changing **n** does not resize the array.

Quantity	Example	Meaning
Capacity	1000	Cells allocated by the declaration.
Active length	4	Values belonging to this input.
Active indices	0 through 3	Cells that this solution should process.

Two boundary checks to understand

For a physical array of capacity **CAPACITY**, element access requires an index in 0 through **CAPACITY** - 1. For the current input, we normally want the smaller range 0 through **n** - 1.

With **n** = 4 and capacity 1000, reading **a[4]** stays inside allocated storage but reads an *inactive* cell. That is a logic error for this input. Reading **a[1000]** goes outside the array itself. Raw array indexing does not automatically perform a bounds check.^{[2-1][2-4]}

The loop you will use most often

for (int i = 0; i < n; ++i) visits exactly the active indices. The condition is **i < n**, not **i <= n**. With **n** = 0, the loop runs zero times.

An input-sized raw array is not standard C++23

Wrong for portable C++23:

```
int n = 0;
std::cin >> n;
int a[n]; // Runtime bound: not a standard C++23 array.
```

Some compilers accept this as a **variable-length-array extension**. That does not make it standard C++23. The bounded-capacity pattern above is the approach used in this session. A runtime-sized **std::vector** is taught in Session 4.^{[2-1][2-5]}

Choose the capacity from the constraints

If the statement guarantees $n \leq 1000$, capacity 1000 is sufficient for zero-based storage. If it allows more, update the capacity; do not quietly ignore the extra input. A built-in array declaration must have a positive bound, so `int a[0];` is not the way to represent an empty input. Keep a positive capacity and use `n = 0` instead.^[2-1]

Large local arrays can exceed an environment's available stack space. These examples keep capacities modest. Memory use grows with both the number of cells and each cell's type. Declaring a huge array "just in case" is not a substitute for reading the constraints.

Human positions and C++ indices

A statement may call the first element "position 1." Translate that to index 0. For a valid one-based position `p`, access `a[p - 1]`. To report an index `i` as a one-based position, print `i + 1`.

Do not subtract one everywhere automatically. Some problems already use zero-based indices. Decide once from the statement and stay consistent.

Pause and predict – 3

An array has capacity 10 and active length 3. Is index 3 part of the input? Is index 10 a valid cell?

Check your prediction. No to both. Index 3 exists in allocated storage but is inactive; index 10 is outside the array. The active indices are 0, 1, and 2.

2.4 Reading and printing an array

A length followed by that many values

The next program reads n , then n integers. For this program, $0 \leq n \leq 1000$, and each integer is between -10^9 and 10^9 . A loop reads directly into each active cell.

Complete program E02 – read a list, then print it

```
#include <iostream>

int main() {
    const int CAPACITY = 1000;
    int a[CAPACITY]{};
    int n = 0;
    std::cin >> n;
    for (int i = 0; i < n; ++i) {
        std::cin >> a[i];
    }
    for (int i = 0; i < n; ++i) {
        if (i > 0) { std::cout << ' '; }
        std::cout << a[i];
    }
    std::cout << '\n';
    return 0;
}
```

Sample input

4
8 -2 5 8

Sample output

8 -2 5 8

Trace the input loop

i	Input read	Array after this read
0	8	[8, 0, 0, 0]
1	-2	[8, -2, 0, 0]
2	5	[8, -2, 5, 0]
3	8	[8, -2, 5, 8]

Only the first four allocated cells are shown. Once i becomes 4, $i < n$ is false and the input loop ends. The output loop begins with a new counter starting at 0.

Why is the space printed before some elements?

`if (i > 0)` prints a separator before every element except the first. This produces one space between values and no leading or trailing space. Finally, `'\n'` ends the output line.

For an empty input list, no values are printed; the final newline still produces an empty line. If a task excludes `n = 0`, you need not invent an output for that case.

Input lines are not array rows

For ordinary numeric input, `>>` skips whitespace. These inputs describe the same one-dimensional list:

```
4
8 -2 5 8
```

```
4 8
-2
5 8
```

The first value is the length, not an element. The next four are stored in the array. The input format tells you their roles.

Printing the array name is not printing the list

For an integer array, `std::cout << a;` does not print its elements. In ordinary expressions the array name commonly converts to a pointer to its first element; output then behaves like pointer output. Use a loop to print integer elements. Pointer details can wait.^[2-6]

Pause and predict – 4

What is wrong with reading using `for (int i = 1; i <= n; ++i)` and then printing indices 0 through `n - 1`?

Check your prediction. The two loops use different ranges. Index 0 was not read, and the final input was stored at index `n`. With zero-based storage, read and print indices 0 through `n - 1`.

2.5 Traversing: sums, counts, minimum, and maximum

A traversal visits the elements

A **traversal** is a loop that visits a sequence. In the usual array loop, i identifies the cell and $a[i]$ is the value stored there. Ask what you need to remember while visiting the cells.

For a total, remember a sum. For “how many?”, remember a count. For a maximum, remember the best value or its index. These are the same running-information patterns from Session 1, now applied to stored values.

Sum of values and count of even values

For E03, $0 \leq n \leq 1000$ and $-10^9 \leq a_i \leq 10^9$. It prints the sum and the number of even elements, in that order.

Complete program E03 – a total and a count

```
#include <iostream>

int main() {
    const int CAPACITY = 1000;
    int a[CAPACITY]{};
    int n = 0;
    std::cin >> n;
    for (int i = 0; i < n; ++i) { std::cin >> a[i]; }
    long long sum = 0;
    int even = 0;
    for (int i = 0; i < n; ++i) {
        sum += a[i];
        if (a[i] % 2 == 0) { ++even; }
    }
    std::cout << sum << ' ' << even << '\n';
    return 0;
}
```

Sample input

4
8 -2 5 8

Sample output

19 3

Index	Value	Sum after visit	Even count
0	8	8	1
1	-2	6	2
2	5	11	2
3	8	19	3

The sum begins at zero because no values have been added. The count begins at zero because no matching elements have been seen. Zero and negative even integers still satisfy $a[i] \% 2 == 0$.

Do not confuse an even index with an even value

$i \% 2 == 0$ asks about the **position**. $a[i] \% 2 == 0$ asks about the **stored number**. In $[5, 2, 7, 8]$, the values at even indices are 5 and 7; the even values are 2 and 8.

A total can need a wider type than one element

Each element might fit in `int`, but 1000 copies of 10^9 total 10^{12} . This is why `sum` is `long long`. If you calculate an average for a nonempty list, use `static_cast<double>(sum) / n` when a fractional result is needed. Never divide by zero for an empty list.

Minimum, maximum, and where they occur

The next program requires $1 \leq n \leq 1000$. It prints the minimum and its **zero-based index**, then the maximum and its index. For a tie, it keeps the first occurrence.

Complete program E04 – remember the best indices

```
#include <iostream>

int main() {
    const int CAPACITY = 1000;
    int a[CAPACITY]{};
    int n = 0;
    std::cin >> n;
    for (int i = 0; i < n; ++i) { std::cin >> a[i]; }
    int minIndex = 0;
    int maxIndex = 0;
    for (int i = 1; i < n; ++i) {
        if (a[i] < a[minIndex]) { minIndex = i; }
        if (a[i] > a[maxIndex]) { maxIndex = i; }
    }
    std::cout << a[minIndex] << ' ' << minIndex << '\n';
    std::cout << a[maxIndex] << ' ' << maxIndex << '\n';
    return 0;
}
```

Sample input

4
-4 -8 -2 -2

Sample output

-8 1
-2 2

Start both indices at 0: the first element is a valid candidate. Begin the scan at index 1 because index 0 is already included. After each iteration, `maxIndex` points to the greatest value seen so far; `minIndex` does the corresponding job for the least value.

The tie rule lives in one comparison

When looking for the maximum, `>` changes the answer only for a strictly larger value, keeping the first maximum. Replacing it with `>=` also updates on ties, keeping the last maximum. Neither is always correct: the statement decides which one you need.

Why not initialize the maximum to zero?

For `[-4, -8, -2]`, zero is not even in the list, but it is greater than every element. Starting with zero would produce the wrong answer. For a nonempty list, initialize from an actual input element.

Pause and predict – 5

For `[6, 9, 9, 4]`, which zero-based index does a maximum scan keep when it updates on `>`? What changes with `>=`?

Check your prediction. Using `>` keeps index 1. Using `>=` keeps index 2, because the later value 9 replaces the earlier tied candidate.

2.6 Searching: a value is not its position

Ask exactly what the problem wants

Searching might mean “does the target appear?”, “where does it first appear?”, “where does it last appear?”, or “how many times does it appear?” These questions use similar loops, but they require different remembered information.

A **linear search** checks the elements in sequence until it finds what it needs or reaches the end. The list does not need to be sorted.

Find the first matching index

For E05, $0 \leq n \leq 1000$; array values and the target are in $[-10^9, 10^9]$. Print the first zero-based matching index, or -1 if there is no match.

Complete program E05 – first occurrence

```
#include <iostream>

int main() {
    const int CAPACITY = 1000;
    int a[CAPACITY]{};
    int n = 0;
    std::cin >> n;
    for (int i = 0; i < n; ++i) { std::cin >> a[i]; }
    int target = 0;
    std::cin >> target;
    int first = -1;
    for (int i = 0; i < n; ++i) {
        if (a[i] == target) {
            first = i;
            break;
        }
    }
    std::cout << first << '\n';
    return 0;
}
```

Sample input

```
4
4 9 4 2
4
```

Sample output

```
0
```

`first = -1` means “not found yet.” It is a **sentinel**: a special value that cannot be a valid index. When a match appears, save `i` and use **break**; a later match must not replace the first one.

A short trace when the answer is absent

For target 7 in [4, 9, 4, 2], each comparison is false. No assignment changes **first**, so it remains -1. The absence of a match is an ordinary outcome, not an error.

Small changes answer different questions

To find the **last** occurrence, remove **break** and keep updating the saved index at every match. To **count** occurrences, start **count** = 0 and increment it at every match; do not stop at the first one.

```
int count = 0;
for (int i = 0; i < n; ++i) {
    if (a[i] == target) { ++count; }
}
```

To ask only about **existence**, a **bool found = false**; flag is enough. Set it to true on a match and stop. Counting one chosen target does not require a frequency array.

Never use the not-found marker as an array index

After searching, check **first != -1** before accessing **a[first]**. Printing **first** is safe; indexing **a[-1]** is not. A sentinel is a signal to your logic, not an extra array cell.

Pause and predict – 6

For [7, 2, 7, 7], give the first index, last index, and count of value 7. What is its first one-based position?

Check your prediction. First index 0, last index 3, count 3, and first one-based position 1. These are four different answers to four different questions.

2.7 Updating, copying, and comparing arrays

An element behaves like an ordinary variable

After checking that `index` is valid, you can assign `a[index] = value;`, increase it with `++a[index];`, or use it in an expression. The assignment changes that one cell only.

A **point update** changes one chosen position. A **conditional update** scans the list and changes every element satisfying a condition.

Replace every occurrence of one value

E06 reads n , the list, then `oldValue` and `newValue`. It requires $0 \leq n \leq 1000$, with all numbers in $[-10^9, 10^9]$, and prints the updated list.

Complete program E06 – conditional replacement

```
#include <iostream>

int main() {
    const int CAPACITY = 1000;
    int a[CAPACITY]{};
    int n = 0;
    std::cin >> n;
    for (int i = 0; i < n; ++i) { std::cin >> a[i]; }
    int oldValue = 0;
    int newValue = 0;
    std::cin >> oldValue >> newValue;
    for (int i = 0; i < n; ++i) {
        if (a[i] == oldValue) { a[i] = newValue; }
    }
    for (int i = 0; i < n; ++i) {
        if (i > 0) { std::cout << ' '; }
        std::cout << a[i];
    }
    std::cout << '\n';
    return 0;
}
```

Sample input

```
4
3 7 3 5
3 9
```

Sample output

```
9 7 9 5
```

When both values are equal, the program still works: matching cells are assigned the value they already contain. When the old value is absent, nothing changes.

Copy elements, not the raw array name

Built-in arrays cannot be assigned to one another with `b = a;`. Copy the active elements with a loop. Similarly, a raw-array comparison is not an element-by-element equality test; compare the lengths and values explicitly.^{[2-6][2-7]}

Complete program E07 – copying creates independent values

```
#include <iostream>

int main() {
    int a[4] = {3, 1, 4, 1};
    int b[4]{};
    for (int i = 0; i < 4; ++i) { b[i] = a[i]; }
    b[0] = 9;
    bool equal = true;
    for (int i = 0; i < 4; ++i) {
        if (a[i] != b[i]) { equal = false; }
    }
    std::cout << a[0] << ' ' << b[0] << '\n';
    std::cout << std::boolalpha << equal << '\n';
    return 0;
}
```

```
3 9
false
```

The loop copies the four current values into `b`. Updating `b[0]` afterwards does not change `a[0]`. The comparison finds a difference, so the arrays' contents are not equal. `std::boolalpha` prints `false` instead of `0`.

Equality includes length and order

For two active lengths `n` and `m`, first require `n == m`. Only then compare corresponding cells. Lists `[1, 2, 3]` and `[1, 3, 2]` contain the same values but are not the same sequence. Lists `[1, 2]` and `[1, 2, 0]` also differ.

Pause and predict – 7

After copying `a` into `b` with a loop, does assigning `a[1] = 99`; change `b[1]` automatically?

Check your prediction. No. Each array has its own cells. The copy loop transferred the values at that time; it did not create a permanent connection between the arrays.

2.8 Neighbours, reversal, and ordered sequences

Adjacent elements are one index apart

Adjacent means next to each other. To compare every element with its predecessor, start at index 1 and use `a[i - 1]`. Starting at index 0 would try to use `a[-1]`.

```
for (int i = 1; i < n; ++i) {  
    // Compare a[i - 1] and a[i].  
}
```

Alternatively, comparing `a[i]` with `a[i + 1]` requires `i + 1 < n`. Write the loop condition from the accesses you plan to make, not from habit.

Printing backwards does not change the array

```
for (int i = n - 1; i >= 0; --i) {  
    if (i < n - 1) { std::cout << ' '; }  
    std::cout << a[i];  
}  
std::cout << '\n';
```

This visits the last element first. The cells remain unchanged. Here `n` and `i` are signed integers; for `n = 0`, `i` starts at `-1` and the loop runs zero times.

Reversing in place changes the stored order

To **reverse** the array, swap the first and last elements, then the second and second-last, and move towards the middle. **In place** means using the existing array rather than a second array of the same length.

The mirror of index `left` is `n - 1 - left`. A temporary variable protects one value while the two assignments exchange the cells.

Complete program E08 – reverse the active array in place

```
#include <iostream>

int main() {
    const int CAPACITY = 1000;
    int a[CAPACITY]{};
    int n = 0;
    std::cin >> n;
    for (int i = 0; i < n; ++i) { std::cin >> a[i]; }
    for (int left = 0; left < n / 2; ++left) {
        int right = n - 1 - left;
        int temporary = a[left];
        a[left] = a[right];
        a[right] = temporary;
    }
    for (int i = 0; i < n; ++i) {
        if (i > 0) { std::cout << ' '; }
        std::cout << a[i];
    }
    std::cout << '\n';
    return 0;
}
```

Sample input

5
10 20 30 40 50

Sample output

50 40 30 20 10

E08 accepts $0 \leq n \leq 1000$, with values in $[-10^9, 10^9]$.

Step	Indices swapped	Array afterwards
1	0 and 4	[50, 20, 30, 40, 10]
2	1 and 3	[50, 40, 30, 20, 10]

There are $n / 2$ swaps. For odd n , the middle element stays where it is. Swapping all n mirrored pairs would undo the reversal by swapping each pair twice.

Is the list already nondecreasing?

Nondecreasing means every next value is at least the previous one. Equal neighbours are allowed: [2, 2, 5] qualifies. **Strictly increasing** requires every next value to be greater, so equal neighbours do not qualify.

Complete program E09 – check the existing order

```
#include <iostream>

int main() {
    const int CAPACITY = 1000;
    int a[CAPACITY]{};
    int n = 0;
    std::cin >> n;
    for (int i = 0; i < n; ++i) { std::cin >> a[i]; }
    bool sorted = true;
    for (int i = 1; i < n; ++i) {
        if (a[i] < a[i - 1]) {
            sorted = false;
            break;
        }
    }
    std::cout << (sorted ? "YES" : "NO") << '\n';
    return 0;
}
```

Sample input**Sample output**

4

YES

2 2 5 9

The same bounds as E08 apply. This program checks order; it does not rearrange anything. An empty or one-element list has no violating adjacent pair, so E09 reports YES.

Pause and predict – 8

Why is `a[left] = a[right]; a[right] = a[left];` not a correct swap?

Check your prediction. The first assignment overwrites the old left value. The second then copies the right value back again, leaving both cells equal. Save the old left value in a temporary variable first.

2.9 Two-dimensional arrays: rows and columns

A table needs two coordinates

A **two-dimensional array** can represent a rectangular table. The first index selects a row; the second selects a column.

```
int a[2][3] = {
    {1, 2, 3},
    {4, 5, 6}
};
```

There are 2 rows, 3 columns, and $2 \times 3 = 6$ integer cells. In C++ terms, this is an array of rows, where each row is itself an array.^[2-1]

	Column 0	Column 1	Column 2
Row 0	1	2	3
Row 1	4	5	6

`a[1][2]` contains 6. `a[2][1]` is invalid: there is no row 2 in this declaration. Swapping the two indices changes what you access; it is not just another spelling of the same cell.

Nested loops describe the traversal

For each row, visit every column. The inner column counter starts at 0 again for each new row. The visits for this example are (0,0), (0,1), (0,2), (1,0), (1,1), and (1,2).

E10 reads the number of rows and columns, followed by the row values. Both dimensions are between 1 and 30; every value is in $[-10^9, 10^9]$.

Complete program E10 – read and print a rectangular table

```

#include <iostream>

int main() {
    const int LIMIT = 30;
    int a[LIMIT][LIMIT]{};
    int rows = 0;
    int columns = 0;
    std::cin >> rows >> columns;
    for (int r = 0; r < rows; ++r) {
        for (int c = 0; c < columns; ++c) {
            std::cin >> a[r][c];
        }
    }
    for (int r = 0; r < rows; ++r) {
        for (int c = 0; c < columns; ++c) {
            if (c > 0) { std::cout << ' '; }
            std::cout << a[r][c];
        }
        std::cout << '\n';
    }
    return 0;
}

```

Sample input

```

2 3
1 2 3
4 5 6

```

Sample output

```

1 2 3
4 5 6

```

There are two active bounds

The declaration reserves `LIMIT` rows of `LIMIT` cells each. The current input uses only `rows` by `columns`. A valid active access requires both `0 <= r && r < rows` and `0 <= c && c < columns`.

With `int a[2][3]`, `a[0][3]` is not a legal shortcut for `a[1][0]`. Respect the bound of the selected row, even though rows are stored consecutively.

A row total and a column total use different loops

To total one row, keep its row index fixed and vary the column. To total one column, keep its column index fixed and vary the row. Reset the running sum when moving to a new row or column.

Complete program E11 – all row sums, then all column sums

```
#include <iostream>

int main() {
    const int LIMIT = 30;
    int a[LIMIT][LIMIT]{};
    int rows = 0;
    int columns = 0;
    std::cin >> rows >> columns;
    for (int r = 0; r < rows; ++r) {
        for (int c = 0; c < columns; ++c) {
            std::cin >> a[r][c];
        }
    }
    for (int r = 0; r < rows; ++r) {
        long long sum = 0;
        for (int c = 0; c < columns; ++c) { sum += a[r][c]; }
        if (r > 0) { std::cout << ' '; }
        std::cout << sum;
    }
    std::cout << '\n';
    for (int c = 0; c < columns; ++c) {
        long long sum = 0;
        for (int r = 0; r < rows; ++r) { sum += a[r][c]; }
        if (c > 0) { std::cout << ' '; }
        std::cout << sum;
    }
    std::cout << '\n';
    return 0;
}
```

Sample input

```
2 3
1 2 3
4 5 6
```

Sample output

```
6 15
5 7 9
```

The dimensions and value bounds are the same as E10. The first row sum is $1 + 2 + 3 = 6$. The first column sum is $1 + 4 = 5$. Each sum is declared *inside* the outer loop so that it starts at zero for the next row or column.

Pause and predict – 9

A matrix has 3 rows and 5 columns. What are its last active row and column indices? How many cells are visited by a full traversal?

Check your prediction. The last row is 2 and the last column is 4. A full traversal visits $3 \times 5 = 15$ cells.

2.10 Square matrices: diagonals and transposition

A square matrix has equal dimensions

An $n \times n$ matrix is square. On its **main diagonal**, row and column indices are equal: $a[i][i]$. On its **other diagonal**, they add up to $n - 1$: $a[i][n - 1 - i]$.

For $n = 3$, the main-diagonal coordinates are (0,0), (1,1), and (2,2). The other-diagonal coordinates are (0,2), (1,1), and (2,0).

Complete program E12 – the two diagonal sums separately

```
#include <iostream>

int main() {
    const int LIMIT = 30;
    int a[LIMIT][LIMIT]{};
    int n = 0;
    std::cin >> n;
    for (int r = 0; r < n; ++r) {
        for (int c = 0; c < n; ++c) { std::cin >> a[r][c]; }
    }
    long long mainSum = 0;
    long long otherSum = 0;
    for (int i = 0; i < n; ++i) {
        mainSum += a[i][i];
        otherSum += a[i][n - 1 - i];
    }
    std::cout << mainSum << ' ' << otherSum << '\n';
    return 0;
}
```

Sample input

```
3
1 2 3
4 5 6
7 8 9
```

Sample output

```
15 15
```

This program requires $1 \leq n \leq 30$, with values in $[-10^9, 10^9]$. Each diagonal has exactly n cells.

Does the centre count once or twice?

For odd n , the two diagonals share the centre. E12 prints *separate* totals, so each total includes it. If a problem asks for the sum of all distinct diagonal cells, count the centre only once. Practice 10 explores that different requirement.

Transposition exchanges the roles of rows and columns

The **transpose** of a table turns each original column into a new row. A 2×3 table becomes 3×2 . To store a transpose in another array, the correspondence would be $b[c][r] = a[r][c]$. To print it, no second array is needed.

Complete program E13 – print the transpose of a rectangle

```
#include <iostream>

int main() {
    const int LIMIT = 30;
    int a[LIMIT][LIMIT]{};
    int rows = 0;
    int columns = 0;
    std::cin >> rows >> columns;
    for (int r = 0; r < rows; ++r) {
        for (int c = 0; c < columns; ++c) {
            std::cin >> a[r][c];
        }
    }
    for (int c = 0; c < columns; ++c) {
        for (int r = 0; r < rows; ++r) {
            if (r > 0) { std::cout << ' '; }
            std::cout << a[r][c];
        }
        std::cout << '\n';
    }
    return 0;
}
```

Sample input

```
2 3
1 2 3
4 5 6
```

Sample output

```
1 4
2 5
3 6
```

The bounds are the same as E10. Each output line fixes a column of the original matrix and visits all its rows. Notice that the program still accesses $a[r][c]$; it changes the order of the loops, not the meaning of the stored coordinates.

Pause and predict – 10

In a 5×5 matrix, what is the other-diagonal column in row 1? Which cell belongs to both diagonals?

Check your prediction. The other-diagonal column is $5 - 1 - 1 = 3$. The shared centre is $a[2][2]$.

2.11 Characters: letters, digits, and single quotes

A character is not a whole string

A `char` can hold one ordinary character such as `'A'`, `'7'`, or `' '`. Use single quotes for a character. Use double quotes for text such as `"A"`, `"007"`, or `"Hello"`.

```
char digitCharacter = '7';
int digitValue = 7;
std::string text = "7"; // Requires #include <string>.
```

These three variables do not have the same type. The first contains a character, the second an integer, and the third a string containing one character. The next topic explains `std::string` in detail.

A digit character can be converted to its numeric value

The characters `'0'` through `'9'` have consecutive character-code values in C++. Therefore, for a character known to be a decimal digit, subtracting `'0'` produces its integer value.^[2-8]

```
char c = '7';
int digit = c - '0'; // The integer 7.
```

Do not subtract the integer 0 instead: that leaves the character's code value, not the decimal digit it represents. Converting a digit value back to a character uses `static_cast<char>('0' + digit)`, provided `digit` is between 0 and 9.

Complete program E14 – recognize and convert one digit

```
#include <iostream>

int main() {
    char c = '0';
    std::cin >> c;
    if ('0' <= c && c <= '9') {
        int digit = c - '0';
        std::cout << digit << ' ' << digit * 2 << '\n';
    } else {
        std::cout << "NOT A DIGIT\n";
    }
    return 0;
}
```

Sample input

7

Sample output

7 14

E14 reads one non-whitespace ASCII character. For a non-digit such as `A`, it prints `NOT A DIGIT`. The range check comes before the conversion's result is treated as a decimal digit.

Recognizing ordinary English letters

For the ASCII input used in this chapter, `'a' <= c && c <= 'z'` recognizes a lowercase letter, and `'A' <= c && c <= 'Z'` recognizes an uppercase letter. These alphabet-range checks rely on the stated ASCII setting; they are not a general Unicode solution.

`'A'` and `'a'` are different characters. A space `' '` and a newline `'\n'` are different too. An escape such as `'\n'` represents one character even though its source spelling uses a backslash and a letter.

Pause and predict – 11

For `char c = '4';`, what does `c - '0'` calculate? Would subtracting `'0'` from `'x'` make sense as a digit conversion?

Check your prediction. It calculates the integer 4. The character `'x'` is not a decimal digit, so first check that the character lies between `'0'` and `'9'`.

2.12 Getting started with `std::string`

Text is a sequence of characters

Include `<string>` and declare a `std::string`. It stores a character sequence whose length can change. Unlike a raw array declaration, it does not require you to choose a fixed capacity in advance.^[2-9]

```
std::string word = "code";
std::string empty;           // Starts with no characters.
std::string spaces(3, ' '); // Three space characters.
```

For `word`, the indices are 0, 1, 2, and 3, containing `'c'`, `'o'`, `'d'`, and `'e'`.

Length, emptiness, and individual characters

`s.size()` and `s.length()` both give the number of stored `char` elements. `s.empty()` tells you whether that number is zero. The dot connects the string to an operation that belongs to it; the parentheses perform the call. For example, write `s.size()`, not `s.size`.^[2-10]

To access an existing character, use `s[i]` with $0 \leq i < \text{length}$. To replace it, assign one character: `s[0] = 'm'`; This changes the content but not the length.

Writing `s[n]` does not append a character

If `n` is the string length, the characters are at indices 0 through `n - 1`. Use `push_back` to append. C++23 gives `s[s.size()]` a special null-terminator rule, but it is not an ordinary content character or a spare append slot. Never assign a non-null character there.^[2-11]

Adding, removing, and clearing characters

`s.push_back(c)` appends one character. `s.pop_back()` removes the last character and requires a nonempty string. `s.clear()` removes all the characters. `s.front()` and `s.back()` access the first and last characters and also require a nonempty string.^{[2-11][2-12]}

Complete program E15 – a string changes step by step

```
#include <iostream>
#include <string>

int main() {
    std::string word = "code";
    std::cout << word.size() << ' ' << word[0] << '\n';
    word[0] = 'm';
    word.push_back('s');
    std::cout << word << '\n';
    word.pop_back();
    std::cout << word << '\n';
    word.clear();
    std::cout << std::boolalpha << word.empty() << '\n';
    return 0;
}
```

```
4 c
modes
mode
true
```

First, `word[0] = 'm'` changes `code` to `mode`. Appending `'s'` creates `modes`; removing the last character brings back `mode`. After clearing, `empty()` is true.

Choose a safe index type for these exercises

String lengths use an unsigned size type rather than `int`. In this chapter, input strings are at most 100000 characters unless a tighter bound is stated, so converting the length to `int` is safe on the training environment's ordinary 32-bit-or-wider `int`.

```
int n = static_cast<int>(s.size());
for (int i = 0; i < n; ++i) {
    // Use s[i].
}
```

Do the conversion *before* subtracting one for a backwards loop. In an empty string, the unsigned expression `s.size() - 1` wraps instead of producing signed `-1`. The technical notes explain this distinction further.

Strings support ordinary value copying

```
std::string a = "cat";
std::string b = a;
b[0] = 'b';
```

Now `a` is still `cat` and `b` is `bat`. Unlike raw arrays, strings support whole-value assignment and content comparisons. They manage their own storage.

An empty string is a valid string

`std::string s;` is initialized and ready to use. It is not like an uninitialized local integer. It is fine to print an empty string or append to it, but there is no first content character to read. Check `!s.empty()` before `s.front()`, `s.back()`, or `s.pop_back()`.

Pause and predict – 12

Starting from `std::string s = "hi";`, what is the result of `s[0] = 'b';` followed by `s.push_back('!');`? What is its length?

Check your prediction. The string becomes `bi!` and has length 3. Replacing the first character preserved the old length; appending increased it by one.

2.13 Reading a word, a line, or a line after a number

Use the input format to choose the tool

`std::cin >> s` reads the next whitespace-delimited token. `std::getline(std::cin, s)` reads a whole line, including spaces, up to the newline. The newline is removed from the input but is not stored in the string.^[2-13]

A **token** here is a run of non-whitespace characters. It might be a word, a code such as **A19**, or punctuation. It is not necessarily an English word.

Complete program E16 – read two tokens

```
#include <iostream>
#include <string>

int main() {
    std::string first;
    std::string second;
    std::cin >> first >> second;
    std::cout << first << '\n' << second << '\n';
    return 0;
}
```

Sample input

HNU ICPC Community

Sample output

HNU
ICPC

Only the first two tokens are read. **Community** remains unread because the program never asks for a third token. Leading whitespace is skipped, and the tokens could be on separate input lines.

Reading a full line preserves its spaces

Complete program E17 – show a line and its length

```
#include <iostream>
#include <string>

int main() {
    std::string line;
    std::getline(std::cin, line);
    std::cout << '[' << line << "]\n";
    std::cout << line.size() << '\n';
    return 0;
}
```

Sample input

HNU ICPC

Sample output

[HNU ICPC]
8

The brackets in the output are deliberately added to show where the line begins and ends. They are not part of the stored line. Input containing two spaces, `hi`, and two more spaces has length 6. An empty line gives an empty string of length 0.

Why can `getline` seem to skip the input?

Consider this **wrong** sequence for a number on one line and a name on the next:

```
int age = 0;
std::string name;
std::cin >> age;
std::getline(std::cin, name); // May read the leftover newline.
```

For input `19` followed by a newline, `>> age` consumes the digits but leaves the line ending. `getline` immediately encounters that newline and finishes with an empty string. It has not ignored your name; it has read the remainder of the current line.

Discard the rest of the number's line once

When the format puts the name on the **next line**, discard the remaining characters on the number's line, including its newline, then call `getline`.

Complete program E18 – a number, then a full name

```
#include <iostream>
#include <string>
#include <limits>

int main() {
    int age = 0;
    std::cin >> age;
    std::cin.ignore(
        std::numeric_limits<std::streamsize>::max(), '\n');
    std::string name;
    std::getline(std::cin, name);
    std::cout << age << '\n';
    std::cout << '[' << name << "]\n";
    return 0;
}
```

Sample input

Sample output

19

19

Mona Ali

[Mona Ali]

The `ignore` call means “discard through the next newline, without a small character-count limit.” Its long limit expression comes from `<limits>`. You can use this pattern before studying templates in depth. It handles spaces after the number as well as the newline.^[2-14]

The fix must match the format

If the number and desired text are on the *same line*, discarding that line would discard useful text. If all fields are tokens, keep using `>>` and no `ignore` is needed. If you call `getline` repeatedly, it already removes each newline; do not call `ignore` before every line.

The shortcut `std::getline(std::cin >> std::ws, s)` discards *all* leading whitespace first. It can be convenient when such whitespace is irrelevant, but it also skips empty lines and leading spaces. Do not use it when those must be preserved.^[2-15]

Pause and predict – 13

A task gives an integer on its own line followed by three text lines, any of which may be empty. Where should you call `ignore`?

Check your prediction. Once after reading the integer with `>>`, to finish that line. Then make three ordinary `getline` calls. Do not skip whitespace or ignore input before each text line.

2.14 Traversing strings: counting and filtering

Use the same loop idea as an array

For a string with length n , an index loop visits $s[0]$ through $s[n - 1]$. At each step, $s[i]$ is a character. You can compare it with `'a'`, test whether it is a digit, or append it to another string.

A local variable such as `char c = s[i];` copies the current character. Assigning a new value to `c` does not change $s[i]$; write to $s[i]$ when you mean to modify the original string.

Count vowels, including uppercase vowels

E19 reads a full ASCII line of length at most 100000. It counts occurrences of the five vowels `a`, `e`, `i`, `o`, `u` in either letter case. Here `y` is not a vowel.

Complete program E19 – count characters that satisfy a condition

```
#include <iostream>
#include <string>

int main() {
    std::string s;
    std::getline(std::cin, s);
    int n = static_cast<int>(s.size());
    int count = 0;
    for (int i = 0; i < n; ++i) {
        char c = s[i];
        bool vowel = c == 'a' || c == 'e' || c == 'i'
                    || c == 'o' || c == 'u' || c == 'A'
                    || c == 'E' || c == 'I' || c == 'O'
                    || c == 'U';
        if (vowel) { ++count; }
    }
    std::cout << count << '\n';
    return 0;
}
```

Sample input

Hello ICPC

Sample output

3

The vowels in the sample are `e`, `o`, and `I`. The long condition is a series of separate character comparisons joined with `||`. It produces one true/false value called `vowel`.

Do not shorten a condition into a different expression

Wrong:

```
if (c == 'a' || 'e') { ++count; }
```

The second part is a nonzero character value, which is treated as true. Write a full comparison for each alternative: `c == 'a' || c == 'e'`. This is the same logical-operator rule from Session 1.

Filtering builds a new sequence

To **filter** text, keep only characters that satisfy a rule. Start with an empty result string and append the characters you want. This preserves their original relative order.

E20 keeps only decimal digits from a full ASCII line of length at most 100000, and also prints the sum of their numeric values.

Complete program E20 – keep digit characters and sum their values

```
#include <iostream>
#include <string>

int main() {
    std::string s;
    std::getline(std::cin, s);
    std::string digits;
    int sum = 0;
    int n = static_cast<int>(s.size());
    for (int i = 0; i < n; ++i) {
        if ('0' <= s[i] && s[i] <= '9') {
            digits.push_back(s[i]);
            sum += s[i] - '0';
        }
    }
    std::cout << digits << '\n' << sum << '\n';
    return 0;
}
```

Sample input

room 20B, floor 3

Sample output

203

5

Digit seen	Result text so far	Numeric sum
'2'	"2"	2
'0'	"20"	2
'3'	"203"	5

`push_back(s[i])` keeps the character, including a leading zero. `s[i] - '0'` converts just that character to a number for addition. These two operations serve different purposes.

Why not convert a very long number first?

A string such as "000123" retains its leading zeros. A string with thousands of digits can exceed every built-in integer type, but summing its digits still only needs a character scan. For at most 100000 digits, the greatest digit sum is $9 \times 100000 = 900000$, which fits in the training environment's `int`.

Do not erase characters while learning the scan pattern

Removing an element shifts later characters and changes indices. A simple `++i` can then skip the next character. Building a separate result string avoids that moving-index problem and makes the intended output easy to trace.

Pause and predict – 14

For input `a09b2`, what digit text and digit sum should E20 produce?

Check your prediction. The text is `092`; the sum is $0 + 9 + 2 = 11$. The leading zero is kept because the result is a string, not a converted integer.

2.15 Joining and comparing strings

Joining text is concatenation

Concatenation means joining sequences end to end. With strings, `+` joins values and `+=` appends to the left-hand string.^{[2-12][2-16]}

```
std::string first = "HNU";
std::string second = "ICPC";
std::string name = first + " " + second;
name += '!'; // name is now "HNU ICPC!".
```

Spaces are not inserted automatically; add them deliberately. Also, `'a' + 'b'` is arithmetic on character values, not the two-character string `"ab"`. To join string literals with `+`, make at least the appropriate starting operand a string, as in `std::string("HNU") + " ICPC"`.

Equality compares the whole content

`a == b` is true when two strings contain exactly the same characters in the same order. Case and spaces matter: `"Code"` differs from `"code"`, and `"hi"` differs from `"hi "`. This is content comparison, unlike a raw integer array name comparison.^[2-17]

Lexicographical order: compare from left to right

Lexicographical order compares the first pair of different characters. If neither differs before one string ends, the shorter one comes first. For our lowercase ASCII words, this behaves like ordinary alphabetical order.^[2-17]

For `"car"` and `"cat"`, `c` and `a` match; `r` comes before `t`, so `"car" < "cat"`. For `"app"` and `"apple"`, the first string ends while all its characters match, so `"app" < "apple"`.

Complete program E21 – compare two tokens, then join them

```

#include <iostream>
#include <string>

int main() {
    std::string a;
    std::string b;
    std::cin >> a >> b;
    if (a < b) { std::cout << "FIRST\n"; }
    else if (a > b) { std::cout << "SECOND\n"; }
    else { std::cout << "EQUAL\n"; }
    std::string joined = a + "-" + b;
    std::cout << joined << '\n';
    return 0;
}

```

Sample input

cat car

Sample output

SECOND

cat-car

E21 reads two nonempty ASCII tokens, each at most 100000 characters. **FIRST** means the first is smaller, **SECOND** means the second is smaller, and **EQUAL** means they match.

Text order is not numeric order

As strings, "10" < "2" because character '1' comes before '2'. As integers, 10 > 2. Decide whether the problem is comparing numbers or comparing text; the visual appearance of the input does not make those operations interchangeable.

Pause and predict – 15

Which is lexicographically smaller: "book" or "books"? Which is smaller: "z" or "aa"?

Check your prediction. "book" is smaller because it is a prefix of "books". "aa" is smaller than "z" because their first characters differ; string length does not decide every comparison.

2.16 Substrings and finding text

A substring is a consecutive piece

A **substring** keeps consecutive characters from a string. For `s = "banana"`, `s.substr(1, 3)` returns `"ana"`: start at index 1 and take 3 characters. The second argument is a **count**, not the final index.^[2-18]

Index	0	1	2	3	4	5
Character	b	a	n	a	n	a

For a piece from index 1 through index `r`, both included, the count is `r - 1 + 1`. For a half-open range `[1, r)`, which excludes `r`, the count is `r - 1`.

What substr does at a boundary

`s.substr(pos)` takes the remaining suffix. A count larger than the remaining length takes only the available characters. `pos == s.size()` is allowed and returns an empty string; `pos > s.size()` reports an out-of-range error. The original string is unchanged.^{[2-18][2-19]}

Find a character or a pattern

`s.find("ana")` returns the first starting position of that pattern. If no match exists, the result is `std::string::npos`. Compare with that named value before treating the result as a position.^[2-20]

Complete program E22 – a substring, a match, and a missing pattern

```
#include <iostream>
#include <string>

int main() {
    std::string s = "banana";
    std::cout << s.substr(1, 3) << '\n';
    auto position = s.find("ana");
    if (position != std::string::npos) {
        std::cout << position << '\n';
    }
    auto missing = s.find("xyz");
    if (missing == std::string::npos) {
        std::cout << "NOT FOUND\n";
    }
    std::cout << s << '\n';
    return 0;
}
```

```
ana
1
NOT FOUND
banana
```

The pattern `ana` appears starting at indices 1 and 3, but the first `find` returns only index 1. The last line shows that neither operation changed `s`.

Why use `auto` for the search result?

`auto position = s.find("ana");` asks the compiler to use the result's actual type. It is a `size/index` type that can also represent `npos`. You do not need to write that type's full name. Avoid first storing it in `int` and then guessing that "not found" is `-1`.

Wrong existence check: `if (s.find("ana"))`. A valid match at index 0 converts to `false`; `npos` is not zero. The correct test is `s.find("ana") != std::string::npos`.

Searching after a chosen start

`s.find("ana", 2)` searches starting at index 2, so in `banana` it finds the occurrence at index 3. Finding repeated or overlapping matches needs a carefully chosen next start; one `find` call does not count all occurrences.

Pause and predict – 16

For `s = "training"`, what is `s.substr(2, 4)`? What result means that `s.find("xyz")` found nothing?

Check your prediction. The substring is `"aini"`, from indices 2, 3, 4, and 5. A missing pattern is signalled by `std::string::npos`.

2.17 Palindromes and reversing text

A palindrome reads the same in both directions

level, noon, and x are palindromes. code is not. Compare each character with its mirror: `s[left]` and `s[n - 1 - left]`.

You only need to compare half of the mirrored pairs. For an odd length, the centre always equals itself. Any mismatch is enough to reject the whole string.

Complete program E23 – check an exact-character palindrome

```
#include <iostream>
#include <string>

int main() {
    std::string s;
    std::getline(std::cin, s);
    int n = static_cast<int>(s.size());
    bool palindrome = true;
    for (int left = 0; left < n / 2; ++left) {
        int right = n - 1 - left;
        if (s[left] != s[right]) {
            palindrome = false;
            break;
        }
    }
    std::cout << (palindrome ? "YES" : "NO") << '\n';
    return 0;
}
```

Sample input

level

Sample output

YES

E23 reads a full ASCII line of length at most 100000, possibly empty. Every stored character matters, including spaces and punctuation. For this task, the empty string counts as a palindrome because it has no mismatched pair.

Trace a positive case and a negative case

For `level`, compare indices 0 and 4: `l == l`. Then compare 1 and 3: `e == e`. The centre is not needed, so the answer stays true.

For `abca`, the outside pair matches, but indices 1 and 2 contain `b` and `c`. Set the flag to false and stop. Matching outside characters alone does not prove that the full string is a palindrome.

Do not silently change the problem's rules

Level is not an exact-character palindrome because 'L' differs from '1'. A task that says “ignore case and spaces” needs a preprocessing step. Without such an instruction, do not delete spaces or change case on your own.

Reversing and checking are different operations

To reverse text *without* changing it, print indices from $n - 1$ down to 0. To reverse the stored string, swap mirrored characters, just as in an array. To check whether it is a palindrome, comparison alone is enough; there is no need to modify it.

```
for (int left = 0; left < n / 2; ++left) {  
    int right = n - 1 - left;  
    char temporary = s[left];  
    s[left] = s[right];  
    s[right] = temporary;  
}
```

This fragment assumes `s` exists and `n` is its current length. It changes `s`; it does not print or return a second string.

Pause and predict – 17

How many mirrored comparisons are needed for a palindrome test on length 6? What about length 5?

Check your prediction. At most 3 for length 6 and 2 for length 5. Both counts are integer division $n / 2$. An early mismatch can stop the scan sooner.

2.18 Counting words: remember whether you are inside one

First define what a word means

For this task, a **word** is a maximal consecutive group of non-space characters. A plain space ' ' is the only separator. Repeated spaces, leading spaces, and trailing spaces do not create extra words. Tabs are outside this task's input format.

For example, " we learn C++ " has three words. Counting spaces and adding one fails because it counts repeated and boundary spaces as if they separated words.

Count the start of each word

Keep a flag `insideWord`. Initially it is false. A space moves you outside a word. A non-space character starts a new word only when you were outside; then increment the count and move inside.

Complete program E24 – word counting with a state flag

```
#include <iostream>
#include <string>

int main() {
    std::string line;
    std::getline(std::cin, line);
    int n = static_cast<int>(line.size());
    int words = 0;
    bool insideWord = false;
    for (int i = 0; i < n; ++i) {
        if (line[i] == ' ') {
            insideWord = false;
        } else if (!insideWord) {
            ++words;
            insideWord = true;
        }
    }
    std::cout << words << '\n';
    return 0;
}
```

Sample input

we learn C++

Sample output

3

In this one sample display, marks a space for visibility; type ordinary spaces as input. The line length is at most 100000, and an empty line is allowed.

Character	Previously inside?	Action	Word count
First space	No	Stay outside	0
Second space	No	Stay outside	0
'w'	No	Start a word	1
'e'	Yes	Continue the word	1
Next space	Yes	Leave the word	1
'l' after spaces	No	Start another word	2

Once inside a word, its second and later characters do not increment the count. The rule counts *beginnings*, not all non-space characters.

A second way to recognize a beginning

Index `i` begins a word exactly when the current character is not a space and either `i == 0` or the previous character is a space. Practice 16 uses that neighbour-based rule, giving a useful second solution to compare with the state flag.

Pause and predict – 18

What should the count be for an empty line, a line of five spaces, and "one two"?

Check your prediction. The answers are 0, 0, and 2. Empty regions between spaces are not words.

2.19 Multiple test cases: reset state, not the wrong input

Each test case is a separate problem instance

A judge may give t cases. Keep per-case variables inside the test-case loop so they are initialized again. If an array is reused, use the current length, not the previous one.

E25 reads $1 \leq t \leq 100$ cases. Each gives $0 \leq n \leq 1000$, followed by n integers in $[-10^9, 10^9]$. It prints one sum per case.

Complete program E25 – several array sums

```
#include <iostream>

int main() {
    const int CAPACITY = 1000;
    int a[CAPACITY]{};
    int t = 0;
    std::cin >> t;
    for (int test = 0; test < t; ++test) {
        int n = 0;
        std::cin >> n;
        for (int i = 0; i < n; ++i) { std::cin >> a[i]; }
        long long sum = 0;
        for (int i = 0; i < n; ++i) { sum += a[i]; }
        std::cout << sum << '\n';
    }
    return 0;
}
```

Sample input

```
3
4
8 9 10 11
1
-7
0
```

Sample output

```
38
-7
0
```

Old cells do not matter when they are inactive

After the second case, some physical cells still contain values from the first case. That is fine: the next calculation visits only the current n cells, and those cells were overwritten by the current input. There is no need to clear all 1000 cells just to sum a newly read list.

However, an array used as *accumulated state* would need suitable resetting. The rule is not “arrays never need clearing”; it is “understand which values the next case will read.”

Several full lines, including empty lines

E38 reads a count on its own line followed by exactly that many lines. It prints each line's length. The count is between 1 and 100, and each line is at most 100000 characters.

Complete program E38 – preserve every text line

```
#include <iostream>
#include <string>
#include <limits>

int main() {
    int t = 0;
    std::cin >> t;
    std::cin.ignore(
        std::numeric_limits<std::streamsize>::max(), '\n');
    for (int test = 0; test < t; ++test) {
        std::string line;
        std::getline(std::cin, line);
        std::cout << line.size() << '\n';
    }
    return 0;
}
```

```
3
Hi

A
```

The three lengths are 2, 0, and 3. There is one **ignore** call after the count, not one before each **getline**. The empty middle line must remain a test case, not be silently skipped.

Pause and predict – 19

A maximum variable is initialized once before all test cases. The first case has maximum 90, and the next contains only -5. Why can the second answer be wrong?

Check your prediction. The old candidate 90 remains and -5 cannot replace it in a greater-than comparison. Initialize the maximum from the first element of *each* nonempty case.

2.20 Worked problems: combine arrays, strings, and conditions

Worked problem A – values above the average

Task. Read a nonempty list and count how many values are strictly greater than its average. Here $1 \leq n \leq 1000$, and $-10^9 \leq a_i \leq 10^9$.

Plan. First read and store the values while finding their total. Only then is the final average known. Make a second pass to classify each stored value. This is a natural reason to use an array.

Avoid unnecessary rounding. If the total is `sum`, the desired condition is $a_i > \text{sum}/n$. Since $n > 0$, multiplying both sides by n gives $a_i n > \text{sum}$. This exact integer comparison avoids both integer-division truncation and floating-point rounding.

Complete program E26 – compare with the average exactly

```
#include <iostream>

int main() {
    const int CAPACITY = 1000;
    int a[CAPACITY]{};
    int n = 0;
    std::cin >> n;
    long long sum = 0;
    for (int i = 0; i < n; ++i) {
        std::cin >> a[i];
        sum += a[i];
    }
    int count = 0;
    for (int i = 0; i < n; ++i) {
        if (1LL * a[i] * n > sum) { ++count; }
    }
    std::cout << count << '\n';
    return 0;
}
```

Sample input

4
2 3 7 8

Sample output

2

The total is 20 and the average is 5. Only 7 and 8 are strictly greater. `1LL` makes the multiplication use `long long` from its first operation; assigning an already-overflowed product to a wider variable would not fix it.

Why it works. The first pass includes every value in the total. The second pass applies an equivalent comparison to every stored element exactly once, so the counter equals the requested number.

Boundary tests. One value gives 0; all-equal values give 0; negative values must work too. For $[-10, -4, -4]$, the average is -6 and two values exceed it. These tests check the condition, not only the arithmetic.

Worked problem B – the common beginning of two strings

Task. Read two lines, each at most 100000 ASCII characters, possibly empty. Print the length of their longest common prefix, then the prefix itself on the next line.

A **prefix** is a piece starting at index 0. Compare the two strings from the beginning. Stop at the first mismatch or when either string ends. No later characters can repair a mismatch in a prefix.

Complete program E27 – find the longest common prefix

```
#include <iostream>
#include <string>

int main() {
    std::string a;
    std::string b;
    std::getline(std::cin, a);
    std::getline(std::cin, b);
    int n = static_cast<int>(a.size());
    int m = static_cast<int>(b.size());
    int length = 0;
    while (length < n && length < m
           && a[length] == b[length]) {
        ++length;
    }
    std::cout << length << '\n';
    std::cout << a.substr(0, length) << '\n';
    return 0;
}
```

Sample input

training
trainers

Sample output

5
train

Trace. The letters t, r, a, i, n match, so `length` reaches 5. The next characters are i and e, which differ. The answer is therefore the first five characters, **train**.

Why the condition is ordered this way. The bounds checks come before `a[length] == b[length]`. Short-circuit `&&` skips the character access as soon as either length boundary is reached. Checking characters first could access beyond the content.

Boundary tests. Two equal strings return their whole length. When one is a prefix of the other, the shorter length is the answer. A mismatch at index 0 or an empty input line gives length 0 and an empty second output line.

Worked problem C – open neighbours in a grid

Task. Read a grid of `.` and `#`, then a valid one-based row and column. Count adjacent `.` cells: up, down, left, and right. Exclude diagonals and the chosen cell itself.

Both dimensions are in $[1, 30]$. Each row is a token of exactly the stated width. The chosen cell may contain either symbol.

Representation. Each string stores a row. `grid[r]` selects that row; `grid[r][c]` selects its character at column `c`.

Complete program E28 – bounds before neighbour access

```
#include <iostream>
#include <string>

int main() {
    const int LIMIT = 30;
    std::string grid[LIMIT];
    int rows = 0;
    int columns = 0;
    std::cin >> rows >> columns;
    for (int r = 0; r < rows; ++r) { std::cin >> grid[r]; }
    int r = 0;
    int c = 0;
    std::cin >> r >> c;
    --r; // Input uses one-based positions.
    --c;
    int open = 0;
    if (r > 0 && grid[r - 1][c] == '.') { ++open; }
    if (r + 1 < rows && grid[r + 1][c] == '.') { ++open; }
    if (c > 0 && grid[r][c - 1] == '.') { ++open; }
    if (c + 1 < columns && grid[r][c + 1] == '.') { ++open; }
    std::cout << open << '\n';
    return 0;
}
```

Sample input

```
3 4
....
.#..
..#.
2 3
```

Sample output

```
2
```

Trace. One-based (2,3) becomes zero-based (1,2). Above is open, below is blocked, left is blocked, and right is open: answer 2.

Why it works. Each of the four conditions checks a neighbour's bounds before its content. The four directions are distinct and exclude the chosen cell.

Boundary tests. A 1×1 grid gives 0. A corner of an all-open 2×2 grid gives 2. A non-square grid helps expose accidentally swapped row and column limits.

2.21 Optional extension: sorting and reversal helpers

Second-reading topic

The core practice can be solved without sorting or iterator notation. This section shows a few standard-library tools after you understand the underlying array and string operations.

Sorting arranges values; searching finds information

To **sort** a sequence is to rearrange its elements into an order. Sorting $[5, 1, 5, -2]$ into nondecreasing order gives $[-2, 1, 5, 5]$. Duplicates remain; sorting does not remove them.

The library tool `std::sort` comes from `<algorithm>`. For the active part of a raw array, use `std::sort(a, a + n);`. This is a call to an already-written library function; learning to write your own functions comes next session.^[2-21]

Complete program E29 – sort only the active array elements

```
#include <iostream>
#include <algorithm>

int main() {
    const int CAPACITY = 1000;
    int a[CAPACITY]{};
    int n = 0;
    std::cin >> n;
    for (int i = 0; i < n; ++i) { std::cin >> a[i]; }
    std::sort(a, a + n);
    for (int i = 0; i < n; ++i) {
        if (i > 0) { std::cout << ' '; }
        std::cout << a[i];
    }
    std::cout << '\n';
    return 0;
}
```

Sample input

4
5 1 5 -2

Sample output

-2 1 5 5

The program accepts $0 \leq n \leq 1000$, with values in $[-10^9, 10^9]$. It changes the stored order, then prints it.

Pause and predict – 20

After `std::sort(a, a + n)`, do the old indices still identify the same original input occurrences?

Check your prediction. Not in general. Sorting rearranges the elements, so a value may move to a different index. Preserve the original order or positions separately when the problem needs them.

The end position is excluded

The two arguments describe a **half-open range**: include the first position and stop before the second. Here `a` marks the first element, and `a + n` marks the position just after the active elements.

`std::sort(a, a + n - 1)` would leave the last active element out of the sort. `std::sort(a, a + CAPACITY)` would include inactive cells. The correct end depends on the active range you intend to process.

Strings provide begin and end positions

`s.begin()` marks the first character, and `s.end()` marks the position after the last character. These position-like objects are called **iterators**. You can pass them to algorithms without learning their implementation now.

Complete program E30 – sorted text, then the reversed sorted text

```
#include <iostream>
#include <string>
#include <algorithm>

int main() {
    std::string s;
    std::getline(std::cin, s);
    std::sort(s.begin(), s.end());
    std::cout << s << '\n';
    std::reverse(s.begin(), s.end());
    std::cout << s << '\n';
    return 0;
}
```

Sample input

baca

Sample output

aabc

cbaa

E30 reads an ASCII line of length at most 100000, possibly empty. For the alphabetic ordering seen in this sample, the input contains lowercase letters. Other ASCII characters sort by the corresponding character order, not by a human-language dictionary.

Reverse does not mean sort descending

`std::reverse` from `<algorithm>` reverses the existing order in place.^[2-22] Applying it to a sorted sequence gives descending order. Applying it to arbitrary text simply reverses that text: reversing `baca` directly gives `acab`, not `cbaa`.

Both `std::sort` and `std::reverse` modify the supplied sequence. Do not try to assign their ordinary `std::` overloads back into the sequence as if they returned a new string.

2.22 Optional extension: shifting, rotating, inserting, and deleting

Second-reading topic

These operations practise assignment order and active length. Trace a four-element example before writing the loop. They are not required by the core practice set.

A right rotation keeps the last value

A one-step right **rotation** changes `[10, 20, 30, 40]` into `[40, 10, 20, 30]`. The last value wraps around to the front. Save it before moving the other elements.

Complete program E31 – rotate the active list right once

```
#include <iostream>

int main() {
    const int CAPACITY = 1000;
    int a[CAPACITY]{};
    int n = 0;
    std::cin >> n;
    for (int i = 0; i < n; ++i) { std::cin >> a[i]; }
    if (n > 0) {
        int last = a[n - 1];
        for (int i = n - 1; i > 0; --i) { a[i] = a[i - 1]; }
        a[0] = last;
    }
    for (int i = 0; i < n; ++i) {
        if (i > 0) { std::cout << ' '; }
        std::cout << a[i];
    }
    std::cout << '\n';
    return 0;
}
```

Sample input

```
4
10 20 30 40
```

Sample output

```
40 10 20 30
```

E31 accepts $0 \leq n \leq 1000$, with values in $[-10^9, 10^9]$. The `n > 0` guard prevents reading a last element when the list is empty.

Assignment direction matters

To move values one place right, start at the right end. First copy index 2 into 3, then 1 into 2, then 0 into 1. Each source is read before being overwritten.

Moving from left to right instead would copy the first value into index 1, then copy that overwritten value into index 2, and so on. You would lose information. A rotation is also different from a shift that discards a value and inserts a default at the other end.

Insertion needs one unused cell

To insert a new value at zero-based index `position`, shift the suffix right and write the new value into the gap. Here $0 \leq n < 1000$, $0 \leq \text{position} \leq n$, and all values lie in $[-10^9, 10^9]$. Position `n` means append after the old last element.

Complete program E32 – insert into a bounded-capacity array

```
#include <iostream>

int main() {
    const int CAPACITY = 1000;
    int a[CAPACITY]{};
    int n = 0;
    std::cin >> n;
    for (int i = 0; i < n; ++i) { std::cin >> a[i]; }
    int position = 0;
    int value = 0;
    std::cin >> position >> value;
    for (int i = n; i > position; --i) { a[i] = a[i - 1]; }
    a[position] = value;
    ++n;
    for (int i = 0; i < n; ++i) {
        if (i > 0) { std::cout << ' '; }
        std::cout << a[i];
    }
    std::cout << '\n';
    return 0;
}
```

Sample input

```
4
10 20 30 40
2 99
```

Sample output

```
10 20 99 30 40
```

The new element makes the active length `n + 1`; the physical capacity is unchanged. The first move writes to old index `n`, which is safe only because `n < CAPACITY` before insertion.

Deletion moves the suffix left

For a nonempty active list and $0 \leq \text{position} \ \&\& \ \text{position} < n$, delete one element as follows:

```
for (int i = position; i + 1 < n; ++i) {
    a[i] = a[i + 1];
}
--n;
```

The old last cell may still contain a value, but it is now outside the active list. There is no need to make that inactive cell disappear. Copying left uses a left-to-right loop so each source is preserved until used.

Trace a deletion before writing its loop

Delete the element at index 2 from [10, 20, 99, 30, 40]. The suffix begins just after the removed value. Move each of those later elements one position left.

Step	Stored cells / active sequence
Before deletion	[10, 20, 99, 30, 40], active length 5.
$i = 2$	Copy $a[3]$ to $a[2]$: [10, 20, 30, 30, 40].
$i = 3$	Copy $a[4]$ to $a[3]$: [10, 20, 30, 40, 40].
Decrease n	Active sequence is now [10, 20, 30, 40]; the final allocated cell is ignored.

Deleting the last active element performs no copying: decreasing n is enough. Deleting the only element leaves active length zero. Insertion and deletion change the active length; reversal and rotation do not.

Pause and predict – 21

An array has capacity 5 and active length 5. Can E32 insert another element without changing its storage strategy?

Check your prediction. No. It would need a sixth cell. The insertion example requires active length strictly less than capacity before the operation.

2.23 Optional extension: subarrays, substrings, and subsequences

Second-reading topic

The names sound similar, but the rules differ. This section is a vocabulary guide and a small two-index exercise; it does not introduce prefix sums or range-query preprocessing.

Consecutive versus allowed to skip

A **subarray** is a consecutive piece of an array. A **substring** is a consecutive piece of a string. A **subsequence** may skip elements, but it preserves the order of the chosen elements.

From `[4, 7, 2, 9]`, `[7, 2]` is a subarray. `[4, 2, 9]` is a subsequence but not a subarray. `[9, 4]` is not a subsequence because it reverses the original order.

From `abcde`, `bcd` is a substring. `ace` is a subsequence but not a substring. A **prefix** begins at the start; a **suffix** reaches the end. Thus `abc` is a prefix and `cde` is a suffix.

An inclusive interval has one more element than its difference

If indices `l` and `r` are both included, its length is $r - l + 1$. If the range is `[l, r)`, the end is excluded and its length is $r - l$. Draw three cells to check a boundary formula rather than memorizing it without a picture.

Check whether one string is a subsequence of another

E33 reads a pattern line and a text line, each at most 100000 ASCII characters. Either may be empty. Decide whether the pattern can be obtained by deleting zero or more text characters without changing the remaining order.

Complete program E33 – one index scans, another counts matches

```

#include <iostream>
#include <string>

int main() {
    std::string pattern;
    std::string text;
    std::getline(std::cin, pattern);
    std::getline(std::cin, text);
    int needed = static_cast<int>(pattern.size());
    int n = static_cast<int>(text.size());
    int matched = 0;
    for (int i = 0; i < n; ++i) {
        if (matched < needed && text[i] == pattern[matched]) {
            ++matched;
        }
    }
    std::cout << (matched == needed ? "YES" : "NO") << '\n';
    return 0;
}

```

Sample input**Sample output**

ace

YES

abcde

Trace the two indices

`i` visits every text character. `matched` tells us how many pattern characters have already been found in order. For `ace` in `abcde`, matching `a` makes `matched = 1`; skip `b`; match `c`; skip `d`; match `e`. All three pattern characters were found.

Using the earliest available match leaves at least as much remaining text as choosing a later match. That is why this left-to-right strategy does not need to undo a successful match. The guard `matched < needed` prevents accessing a pattern character after the whole pattern has already matched.

An empty pattern succeeds immediately in the logical sense: deleting every text character produces it. A nonempty pattern cannot be a subsequence of empty text.

Pause and predict – 22

Is `ac` a substring of `abc`? Is it a subsequence? Is `ca` a subsequence?

Check your prediction. `ac` is not consecutive, so it is not a substring. It is a subsequence. `ca` is not a subsequence because `c` occurs after `a` in the original text.

2.24 Optional extension: other useful array and string tools

Second-reading topic

These are recognition tools, not a new list of things to memorize. Keep using ordinary index loops until they feel comfortable. None of the practice solutions relies on these optional forms.

`std::array`: a fixed-size standard-library wrapper

`std::array<int, 4>` from `<array>` stores exactly four integers. Unlike a raw array, it provides `size()`, whole-array copying, and content comparison. Its size still must be known at compile time; it does not replace a runtime-sized vector.^[2-24]

Complete program E34 – the same fixed collection with `std::array`

```
#include <iostream>
#include <array>

int main() {
    std::array<int, 4> a = {3, 1, 4, 1};
    std::array<int, 4> b = a;
    b[0] = 9;
    std::cout << a.size() << '\n';
    std::cout << a[0] << ' ' << b[0] << '\n';
    std::cout << std::boolalpha << (a == b) << '\n';
    return 0;
}
```

```
4
3 9
false
```

The angle brackets specify the element type and fixed element count. You do not need to study template implementation to read this example. With a partly used `std::array<int, 1000>`, `size()` remains 1000; you would still need an active length for a shorter input.

Range-based for: visit values directly

`for (char c : s)` gives a copy of each character of `s`, one at a time. It is convenient when no index is needed. `for (char& c : s)` gives a writable reference to each actual character; changing `c` changes that character in the string.^[2-23]

Complete program E35 – reading copies, then editing actual characters

```
#include <iostream>
#include <string>

int main() {
    std::string s = "a-b-c";
    for (char c : s) { std::cout << c << ' '; }
    std::cout << '\n';
    for (char& c : s) {
        if (c == '-') { c = '+'; }
    }
    std::cout << s << '\n';
    return 0;
}
```

```
a - b - c
a+b+c
```

The first loop intentionally prints a space after every character. In the second loop, `&` makes `c` another name for the current stored character rather than a copy.

A range loop over a raw array visits its full capacity

With `int a[1000]` and active length 4, `for (int x : a)` still visits all 1000 cells. Use the ordinary index loop for the active range. A range-based loop over a string visits its actual characters.

Insertion, erasure, replacement, and text predicates

String operations can change a whole piece at a time. `s.insert(pos, text)` inserts before `pos`; `s.erase(pos, count)` removes a count of characters; `s.replace(pos, count, text)` replaces that piece. For these position-based forms, a start beyond `s.size()` is out of range.^[2-12]

`starts_with`, `ends_with`, and `contains` ask whether text occurs at the beginning, at the end, or anywhere. The first two are available since C++20, and `contains` is available in C++23. They return a Boolean rather than a match index.^{[2-20][2-30]}

Complete program E36 – modify text, then ask three questions

```
#include <iostream>
#include <string>

int main() {
    std::string s = "team";
    s.insert(0, "HNU ");
    std::cout << s << '\n';
    s.erase(0, 4);
    std::cout << s << '\n';
    s.replace(0, 4, "ICPC");
    std::cout << s << '\n';
    std::cout << std::boolalpha << s.starts_with("IC") << '\n';
    std::cout << s.ends_with("PC") << '\n';
    std::cout << s.contains("CP") << '\n';
    return 0;
}
```

```
HNU team
team
ICPC
true
true
true
```

The start and count arguments refer to the *current* string. After an insertion or erasure, later characters can have different indices. A previously saved length or index may need to be recalculated.

Character classification and changing case

The header `<cctype>` provides tools such as `std::isdigit`, `std::isalpha`, `std::isspace`, `std::tolower`, and `std::toupper`. Character classification and case conversion can depend on the active C locale.^[2-25]

When passing a possibly signed `char` to these functions, convert it to **unsigned char** first. The accepted character argument values are those representable as **unsigned char**, plus the end-of-file marker. This avoids undefined behavior for negative `char` values.^[2-29]

Complete program E37 – lowercase an ASCII line safely

```

#include <iostream>
#include <string>
#include <cctype>

int main() {
    std::string s;
    std::getline(std::cin, s);
    int n = static_cast<int>(s.size());
    for (int i = 0; i < n; ++i) {
        unsigned char c = static_cast<unsigned char>(s[i]);
        s[i] = static_cast<char>(std::tolower(c));
    }
    std::cout << s << '\n';
    return 0;
}

```

Sample input

Hello ICPC 23!

Sample output

hello icpc 23!

E37 reads an ASCII line of at most 100000 characters. The outer conversion to `char` stores the function's integer result back into the string. It is not a complete Unicode case-conversion method.

C-style character arrays are a different representation

```

char letters[3] = {'H', 'N', 'U'}; // Three characters only.
char text[4] = "HNU";             // H, N, U, then '\0'.
std::string s = "HNU";            // Managed string, size 3.

```

A **C-style string** uses a null character `'\0'` to mark its end. The literal initialization of `text` includes that terminator, so four cells are needed.^[2-31] The plain three-character `letters` array has no terminator; do not pass it to output or other operations that expect a null-terminated string.

For this training, prefer `std::string` for text. It tracks its length and manages storage, so you can focus on the problem rather than manually maintaining the terminator and buffer capacity.

Pause and predict – 23

Does `for (char c : s) { c = 'x'; }` replace every character in `s`? What would make it modify the stored characters?

Check your prediction. No; `c` is a copy. Use `char& c` for writable references, or assign to `s[i]` in an index loop.

2.25 Debugging arrays and strings

Start by drawing the cells

Before changing random lines, write the actual length and label every valid index for the smallest failing input. If your loop uses $i + 1$, $i - 1$, or $n - 1 - i$, write those computed indices beside the trace.

A compiler cannot prove that every input keeps your indices valid. A program that happens to run without crashing may still access invalid memory or read inactive cells. Treat correct bounds as part of your algorithm, not as an optional cleanup step.

Common symptoms and likely causes

Symptom	Check first
An extra zero is printed.	Did the output loop visit <code>i == n</code> or all of the capacity?
The first value is missing.	Did one loop start at 1 while the rest used zero-based indices?
All-negative maximum is 0.	Was the initial candidate taken from an input element?
A reversed list is unchanged.	Did you swap every pair twice instead of stopping halfway?
A name becomes empty.	Did <code>getline</code> consume the remainder of a previous number's line?
Every vowel condition is true.	Did you write a comparison on each side of <code> </code> ?
A found pattern is reported absent.	Was index 0 incorrectly treated as false?
Later cases include old values.	Is every calculation using the current length and reset per-case state?

Test cases should target the possible mistake

For an array scan, test one element, all equal values, all negative values, a match at the first or last index, an absent target, and the maximum allowed length. Test an empty input only when the statement permits it.

For matrices, try 1×1 , one row, one column, and a non-square shape. For diagonal problems, compare odd and even dimensions. For text, try one character, an empty line when allowed, repeated characters, mixed case, and leading or repeated spaces.

Use short-circuit checks in the safe order

```
if (i > 0 && a[i - 1] == target) {
    // The previous element exists and matches.
}
```

This fragment assumes `i` is otherwise within the relevant range. The left side prevents `a[-1]` when `i == 0`. Writing the array access first would lose that protection.

Compile with warnings, then inspect the first issue

With a C++23-capable GNU compiler, a useful command is:^{[2-26][2-27]}

```
g++ -std=c++23 -Wall -Wextra -Wpedantic main.cpp -o main
```

Adding `-pedantic-errors` rejects diagnosed non-standard extensions such as the variable-length raw arrays discussed earlier. Warnings about signed/unsigned comparison often point to a string-length mismatch; understand the types rather than hiding the warning with an unexplained cast.

Optional runtime checks are a second line of defence

Where supported, GCC's undefined-behavior and address sanitizers can detect many invalid operations during a run. Standard-library assertion settings can help too. They do not replace boundary reasoning and they only examine executed paths. The source package includes the exact flags used for its verification tests.^[2-28]

Pause and predict – 24

A program works on three values but fails when there is only one. Which accesses deserve immediate attention?

Check your prediction. Look for `a[1]`, `a[i - 1]`, `a[i + 1]`, and assumptions that a second element exists. Trace the one-element case directly; the smallest input often reveals the missing guard.

2.26 Practice problems

Try the problems without copying a worked program. Write the input format, the meaning of each variable, and a tiny trace before typing. If you get stuck, use the corresponding hint in Section 2.27 before opening the solution.

Shared conventions

For **Problems 1–8**, $1 \leq n \leq 1000$ and every array value is in $[-10^9, 10^9]$. Problem 6 has two arrays of that same length. Later problems state their own bounds. All inputs satisfy their stated format.

An input position is zero-based unless the problem explicitly says **one-based**. YES and NO are uppercase. A visible-space symbol `␣` in a sample stands for an ordinary space, not a character you should type.

Suggested route. Start with 1, 2, 3, 5, 11, and 12. Continue with the remaining array and string tasks, then finish with 19 and 20. The optional-extension sections are not prerequisites for this practice.

1. Print in reverse order

Print the array from its last element to its first. You do not have to change the stored array.

Input. Read n , followed by n integers.

Output. Print the values in reverse order on one line, separated by one space.

Sample input	Sample output
4 2 7 1 8	8 1 7 2

2. Sum the values at even indices

Find the sum of the elements at zero-based indices 0, 2, 4, and so on. This is about indices, not whether the values are even.

Input. Read n , followed by n integers.

Output. Print one integer: the required sum.

Sample input	Sample output
5 5 2 7 8 3	15

3. First one-based position

Find the first occurrence of a target and report its one-based position. Report -1 when it is absent.

Input. Read n , the n values, then **target**, which is in $[-10^9, 10^9]$.

Output. Print the first one-based position, or -1 .

Sample input	Sample output
4 5 8 5 2 5	1

4. The last occurrence of the maximum

Find the maximum value. If it appears more than once, report the last occurrence.

Input. Read n , followed by n integers.

Output. Print the maximum and its last one-based position on one line.

Sample input

Sample output

4

-1 3

-3 -1 -1 -5

5. Replace negative values with zero

Change every negative array value to zero. Keep all nonnegative values and keep the same number of elements.

Input. Read n , followed by n integers.

Output. Print the updated array on one line.

Sample input

Sample output

4

0 0 5 0

-2 0 5 -1

6. Do the arrays match?

Decide whether two arrays of the same length have equal values at every corresponding index.

Input. Read n , then all n values of array a , then all n values of array b .

Output. Print YES if the sequences are identical; otherwise print NO.

Sample input

Sample output

3

NO

1 2 3

1 3 2

7. A symmetric array

An array is symmetric when it reads the same from left to right and from right to left. Decide whether the given array is symmetric.

Input. Read n , followed by n integers.

Output. Print YES or NO.

Sample input

Sample output

5

YES

1 2 9 2 1

8. Count strict rises

Count how many elements are strictly greater than the element immediately before them. The first element has no predecessor.

Input. Read n , followed by n integers.

Output. Print the number of strict rises.

Sample input	Sample output
5 2 5 5 1 4	2

9. One sum for each row

Calculate the sum of every row of a rectangular matrix.

Input. Read `rows` and `columns`, both in $[1, 30]$, then the matrix in row order. Each value lies in $[-10^9, 10^9]$.

Output. Print one row sum per line, from the first row to the last.

Sample input	Sample output
2 3 1 2 3 4 5 6	6 15

10. Distinct cells on the two diagonals

Sum all cells that belong to either diagonal of a square matrix. Count each cell only once, including the centre when the diagonals meet.

Input. Read n , where $1 \leq n \leq 30$, then the $n \times n$ matrix. Each value lies in $[-10^9, 10^9]$.

Output. Print the sum of the distinct diagonal cells.

Sample input	Sample output
3 1 2 3 4 5 6 7 8 9	25

11. Count one chosen letter

Count the occurrences of a chosen lowercase letter in a word.

Input. Read a nonempty lowercase ASCII string `s` of length at most 100000, then one lowercase character `target`.

Output. Print the number of matches.

Sample input	Sample output
banana a	3

12. Digit sum of a very long number

Calculate the sum of all decimal digits in a string. Leading zeros are allowed, and the represented number may be too large for an integer type.

Input. Read one token containing between 1 and 100000 decimal digits.

Output. Print its digit sum as an integer.

Sample input	Sample output
009812	20

13. Remove lowercase vowels

Remove every a, e, i, o, and u from a lowercase word. Keep the order of all other characters.

Input. Read a nonempty lowercase ASCII string of length at most 100000.

Output. Print the remaining text, followed by a newline. The result may be empty.

Sample input	Sample output
education	dctn

14. Palindrome word

Decide whether a lowercase word is identical to its reversal.

Input. Read a nonempty lowercase ASCII string of length at most 100000.

Output. Print YES if it is a palindrome; otherwise print NO.

Sample input	Sample output
noon	YES

15. Print the lexicographically smaller word

Given two lowercase words, print the one that comes first in lexicographical order. If they are equal, print that word once.

Input. Read two nonempty lowercase ASCII strings, each at most 100000 characters.

Output. Print the smaller string.

Sample input	Sample output
app apple	app

16. Count words in a full line

Count maximal groups of non-space characters. Plain spaces are the only separators; repeated, leading, and trailing spaces are allowed.

Input. Read one full ASCII line, containing no tabs, of length at most 100000. The line may be empty.

Output. Print the number of words.

Sample input	Sample output
we solve problems	3

17. Extract a piece of text

Print a substring beginning at a given zero-based index and containing a given number of characters.

Input. Read a nonempty lowercase ASCII string **s**, then integers **start** and **count**. Its length is at most 100000; $0 \leq \text{start} \leq |s|$ and $0 \leq \text{count} \leq |s| - \text{start}$.

Output. Print exactly the requested substring, followed by a newline.

Sample input	Sample output
training 2 4	aini

18. Length of the common prefix

Find how many characters match consecutively from the beginning of two strings.

Input. Read two full lines. Each contains zero to 100000 lowercase ASCII letters; an empty line is allowed.

Output. Print the length of the longest common prefix.

Sample input	Sample output
train training	5

19. Independent maximums for several cases

Find a separate maximum for each test case. Values from earlier cases must not affect later answers.

Input. Read $1 \leq t \leq 100$. Each case gives $1 \leq n \leq 1000$, then **n** integers in $[-10^9, 10^9]$.

Output. Print one maximum per case, each on its own line.

Sample input	Sample output
3	9
3	-5
7 9 2	-4
1	
-5	
2	
-4 -8	

20. Check a simple text rule

A token is accepted when it has at least 8 characters and contains at least one uppercase letter, one lowercase letter, and one digit. This is a programming exercise, not a password-security recommendation.

Input. Read one token of length 1 to 100000, containing only ASCII letters and decimal digits.

Output. Print YES if all four requirements hold; otherwise print NO.

Sample input	Sample output
HnuCode23	YES

2.27 Hints before solutions

Read only the hint for the problem you are currently attempting. Return to your code and test the idea before opening the full solution.

- 1. Print in reverse order** After reading the array, where is the last element? Make a signed counter move downwards. Explained solution.
- 2. Sum the values at even indices** Start the index at zero and increase it by two. The element values may be negative or odd. Explained solution.
- 3. First one-based position** Scan left to right. On a match, convert index i to position $i + 1$ and stop. Explained solution.
- 4. The last occurrence of the maximum** Keep an index of the best value. Should an equal value update it? Explained solution.
- 5. Replace negative values with zero** Visit each cell and assign zero only when the stored value is less than zero. Explained solution.
- 6. Do the arrays match?** Start with a true flag. A single mismatch is enough to make the answer false. Explained solution.
- 7. A symmetric array** Pair index i with $n - 1 - i$. How many different mirrored pairs exist? Explained solution.
- 8. Count strict rises** Start at index 1 and compare $a[i]$ with $a[i - 1]$. Equality is not a rise. Explained solution.
- 9. One sum for each row** Keep the row fixed while scanning its columns. Where should the sum be reset? Explained solution.
- 10. Distinct cells on the two diagonals** Visit each cell once. Include it when $r == c$ or $r + c == n - 1$. Explained solution.

Hints 11–20

- 11. Count one chosen letter** Compare each character with the target. Do not stop after the first match. Explained solution.
- 12. Digit sum of a very long number** Keep the input as text. Convert one digit character at a time with subtraction of '0'. Explained solution.
- 13. Remove lowercase vowels** Build an empty result string. Append the current character only when it is not one of the five vowels. Explained solution.
- 14. Palindrome word** Compare mirrored characters. Stop at half the length, not at the end. Explained solution.
- 15. Print the lexicographically smaller word** String comparison already implements the required left-to-right rule. Comparing only lengths is not enough. Explained solution.
- 16. Count words in a full line** A word begins at a non-space character whose predecessor is a space, or at index 0 when it is non-space. Explained solution.
- 17. Extract a piece of text** The two arguments to `substr` already mean start position and character count. Explained solution.
- 18. Length of the common prefix** Advance while both strings still have a character and the current pair matches. Check both lengths first. Explained solution.
- 19. Independent maximums for several cases** Read the current array, then initialize the current maximum from its own first element. Explained solution.
- 20. Check a simple text rule** Keep three separate Boolean flags for the character categories, then combine them with the length condition. Explained solution.

2.28 Explained practice solutions

These are complete programs, but reading a finished program is not the same as being able to write it. First explain the plan. Then trace the sample, type the program, and create a test aimed at its most likely mistake.

A quick way to review your own attempt

Compare the meaning of your variables with the solution, not only their names. Check the first and last iteration. Identify why every array or string access is valid. Confirm that ties, missing targets, empty results, and repeated cases follow the statement.

Solution index

No.	Explained solution
1	Print in reverse order
2	Sum the values at even indices
3	First one-based position
4	The last occurrence of the maximum
5	Replace negative values with zero
6	Do the arrays match?
7	A symmetric array
8	Count strict rises
9	One sum for each row
10	Distinct cells on the two diagonals
11	Count one chosen letter
12	Digit sum of a very long number
13	Remove lowercase vowels
14	Palindrome word
15	Print the lexicographically smaller word
16	Count words in a full line
17	Extract a piece of text
18	Length of the common prefix
19	Independent maximums for several cases
20	Check a simple text rule

Each solution starts on a fresh page. The programs are also included separately under `practice/session_02/` in the source package.

Solution 1 – Print in reverse order

Plan. Store the list so every value remains available. Start the output index at $n - 1$, print that cell, and decrease the index until it becomes negative. Print separators between the output values, not before the first one.

Complete solution – C++23

```
#include <iostream>

int main() {
    const int CAPACITY = 1000;
    int a[CAPACITY]{};
    int n = 0;
    std::cin >> n;
    for (int i = 0; i < n; ++i) { std::cin >> a[i]; }
    for (int i = n - 1; i >= 0; --i) {
        if (i < n - 1) { std::cout << ' '; }
        std::cout << a[i];
    }
    std::cout << '\n';
    return 0;
}
```

Sample input	Sample output
4	8 1 7 2
2 7 1 8	

Trace the sample. The input [2, 7, 1, 8] is visited at indices 3, 2, 1, and 0. The resulting output is 8 1 7 2; the array itself is unchanged.

Why it works. The descending counter visits every valid index exactly once in the opposite order, so no element is skipped or duplicated.

Tests and common mistakes. One value must print once. Try an even length, an odd length, and repeated values. The task guarantees that the list is nonempty.

Solution 2 – Sum the values at even indices

Plan. Use the indices themselves to choose the cells: 0, 2, 4, and so on, while the index stays below n . Add the value in each selected cell to a `long long` total.

Complete solution – C++23

```
#include <iostream>

int main() {
    const int CAPACITY = 1000;
    int a[CAPACITY]{};
    int n = 0;
    std::cin >> n;
    for (int i = 0; i < n; ++i) { std::cin >> a[i]; }
    long long sum = 0;
    for (int i = 0; i < n; i += 2) { sum += a[i]; }
    std::cout << sum << '\n';
    return 0;
}
```

Sample input

Sample output

5

15

5 2 7 8 3

Trace the sample. For [5, 2, 7, 8, 3], select values 5, 7, and 3. Their total is 15. Selecting the even values 2 and 8 would answer a different question.

Why it works. The loop begins at the first even nonnegative index and advances by two, so it visits all and only the requested indices.

Tests and common mistakes. At length 1, include index 0. Negative values still contribute. Up to 500 selected values of size 10^9 can total 5×10^{11} , so use a wide total.

Solution 3 – First one-based position

Plan. Initialize the answer to -1. For each cell, compare its value with the target. On the first match, store $i + 1$ and break. Saving a position directly avoids converting the absent sentinel later.

Complete solution – C++23

```
#include <iostream>

int main() {
    const int CAPACITY = 1000;
    int a[CAPACITY]{};
    int n = 0;
    std::cin >> n;
    for (int i = 0; i < n; ++i) { std::cin >> a[i]; }
    int target = 0;
    std::cin >> target;
    int answer = -1;
    for (int i = 0; i < n; ++i) {
        if (a[i] == target) {
            answer = i + 1;
            break;
        }
    }
    std::cout << answer << '\n';
    return 0;
}
```

Sample input

```
4
5 8 5 2
5
```

Sample output

```
1
```

Trace the sample. For [5, 8, 5, 2] and target 5, the first match is at index 0. The requested position is therefore 1, not 0.

Why it works. The scan visits earlier indices first. Breaking at the first match guarantees no later occurrence can replace it; if no match exists, the sentinel remains unchanged.

Tests and common mistakes. Test a match at the first cell, at the last cell, repeated matches, and no match. Never use the -1 answer as an array index.

Solution 4 – The last occurrence of the maximum

Plan. Use index 0 as the initial candidate because the input is nonempty. Replace the candidate when the current value is greater than *or equal to* the saved maximum. Print the saved value and index plus one.

Complete solution – C++23

```
#include <iostream>

int main() {
    const int CAPACITY = 1000;
    int a[CAPACITY]{};
    int n = 0;
    std::cin >> n;
    for (int i = 0; i < n; ++i) { std::cin >> a[i]; }
    int best = 0;
    for (int i = 1; i < n; ++i) {
        if (a[i] >= a[best]) { best = i; }
    }
    std::cout << a[best] << ' ' << best + 1 << '\n';
    return 0;
}
```

Sample input

4
-3 -1 -1 -5

Sample output

-1 3

Trace the sample. For $[-3, -1, -1, -5]$, the candidate first moves to index 1, then to index 2 on the tie. The output is -1 3.

Why it works. After each visit, the saved index is the last occurrence of the greatest value among the cells already visited. That property remains true when a smaller, larger, or equal value arrives.

Tests and common mistakes. All-negative values expose a wrong initial maximum of zero. All-equal values should select the final position. A one-element input selects position 1.

Solution 5 – Replace negative values with zero

Plan. Read the whole list, then scan it. When $a[i] < 0$, write zero into that cell. Finally print every active cell. This is replacement, not filtering: no element is removed.

Complete solution – C++23

```
#include <iostream>

int main() {
    const int CAPACITY = 1000;
    int a[CAPACITY]{};
    int n = 0;
    std::cin >> n;
    for (int i = 0; i < n; ++i) { std::cin >> a[i]; }
    for (int i = 0; i < n; ++i) {
        if (a[i] < 0) { a[i] = 0; }
    }
    for (int i = 0; i < n; ++i) {
        if (i > 0) { std::cout << ' '; }
        std::cout << a[i];
    }
    std::cout << '\n';
    return 0;
}
```

Sample input	Sample output
4	0 0 5 0
-2 0 5 -1	

Trace the sample. $[-2, 0, 5, -1]$ becomes $[0, 0, 5, 0]$. The original zero remains zero and the positive value stays unchanged.

Why it works. Every element is visited once. The condition changes exactly the negative values and leaves every other value alone.

Tests and common mistakes. Test all negative, all positive, and all zero values. The output must always contain exactly n elements, even when all are zero.

Solution 6 – Do the arrays match?

Plan. Store both arrays. Begin with `same = true`. Compare corresponding elements; any mismatch makes the flag false. Do not change it back to true after a later matching pair.

Complete solution – C++23

```
#include <iostream>

int main() {
    const int CAPACITY = 1000;
    int a[CAPACITY]{};
    int b[CAPACITY]{};
    int n = 0;
    std::cin >> n;
    for (int i = 0; i < n; ++i) { std::cin >> a[i]; }
    for (int i = 0; i < n; ++i) { std::cin >> b[i]; }
    bool same = true;
    for (int i = 0; i < n; ++i) {
        if (a[i] != b[i]) { same = false; }
    }
    std::cout << (same ? "YES" : "NO") << '\n';
    return 0;
}
```

Sample input	Sample output
3	NO
1 2 3	
1 3 2	

Trace the sample. [1, 2, 3] and [1, 3, 2] have the same values in different positions, so they are not identical sequences.

Why it works. The flag remains true exactly when no differing pair has been found. If every pair matches, the entire same-length sequence matches.

Tests and common mistakes. Test identical arrays, only the first cell different, and only the last cell different. A raw-array-name comparison does not perform this content test.

Solution 7 – A symmetric array

Plan. Start with a true flag and compare each element in the first half with its mirror in the second half. A mismatch makes the flag false. There is no need to reverse or copy the array.

Complete solution – C++23

```
#include <iostream>

int main() {
    const int CAPACITY = 1000;
    int a[CAPACITY]{};
    int n = 0;
    std::cin >> n;
    for (int i = 0; i < n; ++i) { std::cin >> a[i]; }
    bool symmetric = true;
    for (int i = 0; i < n / 2; ++i) {
        if (a[i] != a[n - 1 - i]) { symmetric = false; }
    }
    std::cout << (symmetric ? "YES" : "NO") << '\n';
    return 0;
}
```

Sample input

5
1 2 9 2 1

Sample output

YES

Trace the sample. For [1, 2, 9, 2, 1], the pairs are 1/1 and 2/2; the middle 9 has no separate partner to check.

Why it works. Every distinct mirrored pair is checked. Symmetry means precisely that all those pairs agree, so the final flag answers the definition directly.

Tests and common mistakes. One element is symmetric. Test an even-length symmetric array, an odd-length one, and a mismatch close to the middle rather than only at the ends.

Solution 8 – Count strict rises

Plan. Initialize the count to zero. For every index from 1 through $n - 1$, increment the count exactly when the current value exceeds its immediate predecessor.

Complete solution – C++23

```
#include <iostream>

int main() {
    const int CAPACITY = 1000;
    int a[CAPACITY]{};
    int n = 0;
    std::cin >> n;
    for (int i = 0; i < n; ++i) { std::cin >> a[i]; }
    int rises = 0;
    for (int i = 1; i < n; ++i) {
        if (a[i] > a[i - 1]) { ++rises; }
    }
    std::cout << rises << '\n';
    return 0;
}
```

Sample input**Sample output**

5

2

2 5 5 1 4

Trace the sample. In $[2, 5, 5, 1, 4]$, the rises are $2 \rightarrow 5$ and $1 \rightarrow 4$. The equal pair $5 \rightarrow 5$ is not counted.

Why it works. Each adjacent pair has exactly one right-hand index. The loop visits each such index once and counts exactly the pairs satisfying the strict inequality.

Tests and common mistakes. One element gives 0. An all-equal list gives 0. A strictly increasing list of length n gives $n - 1$, not n .

Solution 9 – One sum for each row

Plan. Read the matrix with nested loops. Then use one outer iteration per row. Create a fresh `long long sum = 0` inside that iteration and add the row's columns before printing.

Complete solution – C++23

```
#include <iostream>

int main() {
    const int LIMIT = 30;
    int a[LIMIT][LIMIT]{};
    int rows = 0;
    int columns = 0;
    std::cin >> rows >> columns;
    for (int r = 0; r < rows; ++r) {
        for (int c = 0; c < columns; ++c) {
            std::cin >> a[r][c];
        }
    }
    for (int r = 0; r < rows; ++r) {
        long long sum = 0;
        for (int c = 0; c < columns; ++c) { sum += a[r][c]; }
        std::cout << sum << '\n';
    }
    return 0;
}
```

Sample input

```
2 3
1 2 3
4 5 6
```

Sample output

```
6
15
```

Trace the sample. The rows [1, 2, 3] and [4, 5, 6] produce 6 and 15 on separate lines. The second total must not include the first.

Why it works. For a fixed row, the inner loop adds every column exactly once. Reinitializing the sum ensures each output describes only its own row.

Tests and common mistakes. Test one row, one column, and a non-square matrix. A row of 30 values equal to 10^9 totals 3×10^{10} , so use `long long`.

Solution 10 – Distinct cells on the two diagonals

Plan. Traverse the whole matrix. Use one `if` statement joined by `||` to recognize membership in either diagonal. Add the cell once whenever that single condition is true.

Complete solution – C++23

```
#include <iostream>

int main() {
    const int LIMIT = 30;
    int a[LIMIT][LIMIT]{};
    int n = 0;
    std::cin >> n;
    for (int r = 0; r < n; ++r) {
        for (int c = 0; c < n; ++c) { std::cin >> a[r][c]; }
    }
    long long sum = 0;
    for (int r = 0; r < n; ++r) {
        for (int c = 0; c < n; ++c) {
            if (r == c || r + c == n - 1) { sum += a[r][c]; }
        }
    }
    std::cout << sum << '\n';
    return 0;
}
```

Sample input

```
3
1 2 3
4 5 6
7 8 9
```

Sample output

```
25
```

Trace the sample. For the matrix containing 1 through 9 in row order, the included values are 1, 3, 5, 7, and 9. Their total is 25, not 30.

Why it works. Every matrix cell is visited only once. The Boolean condition includes exactly diagonal cells; even when both parts are true at the centre, the body runs once.

Tests and common mistakes. For $n = 1$, the only cell is counted once. For even n , there is no shared centre. Negative diagonal values still belong in the total.

Solution 11 – Count one chosen letter

Plan. Read the word and target. Visit each valid string index. When `s[i] == target`, increase the counter. The target is a `char`, so one-character comparisons are appropriate.

Complete solution – C++23

```
#include <iostream>
#include <string>

int main() {
    std::string s;
    char target = 'a';
    std::cin >> s >> target;
    int n = static_cast<int>(s.size());
    int count = 0;
    for (int i = 0; i < n; ++i) {
        if (s[i] == target) { ++count; }
    }
    std::cout << count << '\n';
    return 0;
}
```

Sample input

banana a

Sample output

3

Trace the sample. In `banana`, the letter `a` appears at indices 1, 3, and 5, giving a count of 3.

Why it works. Each occurrence contributes one increment, and no nonmatching character contributes. After the scan, the total therefore equals the number of occurrences.

Tests and common mistakes. Test a missing letter, a one-character word, and a word consisting entirely of the target. This task does not need a frequency array.

Solution 12 – Digit sum of a very long number

Plan. Store the input in a string, not an integer. The input guarantee makes every character a decimal digit. Subtract '0' from each character and add the resulting digit value.

Complete solution – C++23

```
#include <iostream>
#include <string>

int main() {
    std::string digits;
    std::cin >> digits;
    int n = static_cast<int>(digits.size());
    int sum = 0;
    for (int i = 0; i < n; ++i) { sum += digits[i] - '0'; }
    std::cout << sum << '\n';
    return 0;
}
```

Sample input	Sample output
009812	20

Trace the sample. 009812 contributes $0 + 0 + 9 + 8 + 1 + 2 = 20$. The leading zeros are read but contribute nothing to the sum.

Why it works. Each digit is converted and added exactly once. The result does not depend on whether the whole represented number fits in a numeric type.

Tests and common mistakes. All zeros give 0. At the maximum length, all nines give 900000. This bound fits in the training environment's `int`.

Solution 13 – Remove lowercase vowels

Plan. For each character, compute whether it is one of the five named vowels. Append it to **result** only when that condition is false. Building a separate result avoids index shifts during the scan.

Complete solution – C++23

```
#include <iostream>
#include <string>

int main() {
    std::string s;
    std::cin >> s;
    std::string result;
    int n = static_cast<int>(s.size());
    for (int i = 0; i < n; ++i) {
        char c = s[i];
        bool vowel = c == 'a' || c == 'e' || c == 'i'
                    || c == 'o' || c == 'u';
        if (!vowel) { result.push_back(c); }
    }
    std::cout << result << '\n';
    return 0;
}
```

Sample input	Sample output
education	dctn

Trace the sample. In education, keep d, c, t, and n. The answer is dctn.

Why it works. The scan visits characters in their original order. It appends exactly the non-vowels, so the result preserves their relative order and excludes every vowel.

Tests and common mistakes. An all-vowel word produces an empty output line. A word without vowels stays unchanged. Uppercase input is outside this task's constraints.

Solution 14 – Palindrome word

Plan. Convert the bounded length to `int`. Compare each character in the first half with the corresponding character counted from the end. Keep a flag recording whether any pair differs.

Complete solution – C++23

```
#include <iostream>
#include <string>

int main() {
    std::string s;
    std::cin >> s;
    int n = static_cast<int>(s.size());
    bool palindrome = true;
    for (int i = 0; i < n / 2; ++i) {
        if (s[i] != s[n - 1 - i]) { palindrome = false; }
    }
    std::cout << (palindrome ? "YES" : "NO") << '\n';
    return 0;
}
```

Sample input

noon

Sample output

YES

Trace the sample. For noon, indices 0 and 3 match, and indices 1 and 2 match. Both distinct mirrored pairs agree.

Why it works. A sequence equals its reversal exactly when all mirrored positions contain the same character. The loop checks every distinct such pair.

Tests and common mistakes. A one-letter word is a palindrome. Test odd and even lengths and a failure in an inner pair. No case conversion is necessary because input is already lowercase.

Solution 15 – Print the lexicographically smaller word

Plan. Read both strings and compare them with `<=`. Print the first when it is smaller or equal; otherwise print the second. No sorting algorithm is needed for two values.

Complete solution – C++23

```
#include <iostream>
#include <string>

int main() {
    std::string a;
    std::string b;
    std::cin >> a >> b;
    if (a <= b) { std::cout << a << '\n'; }
    else { std::cout << b << '\n'; }
    return 0;
}
```

Sample input	Sample output
app apple	app

Trace the sample. `app` comes before `apple`, because the shorter word ends while all of its characters match the beginning of the longer one.

Why it works. The standard string comparison implements lexicographical order. Choosing the operand on the smaller-or-equal side therefore prints exactly the required word.

Tests and common mistakes. Test equal strings, a prefix pair, and a difference in the first character. For `z` and `aa`, the answer is `aa` despite its greater length.

Solution 16 – Count words in a full line

Plan. Use `getline` to retain spaces. Count an index exactly when its character is not a space and it is either index 0 or follows a space. The parenthesized `||` avoids checking index `-1`.

Complete solution – C++23

```
#include <iostream>
#include <string>

int main() {
    std::string line;
    std::getline(std::cin, line);
    int n = static_cast<int>(line.size());
    int count = 0;
    for (int i = 0; i < n; ++i) {
        if (line[i] != ' ' && (i == 0 || line[i - 1] == ' ')) {
            ++count;
        }
    }
    std::cout << count << '\n';
    return 0;
}
```

Sample input

we solve problems

Sample output

3

Trace the sample. In " we solve problems ", the starts are the first letters of we, solve, and problems. Extra spaces add no starts.

Why it works. Every maximal non-space group has one first character, and the condition recognizes precisely that character. Counting these starts counts the words one-for-one.

Tests and common mistakes. An empty line and an all-space line both give 0. One long token gives 1. This previous-character solution and E24's state-flag solution solve the same task.

Solution 17 – Extract a piece of text

Plan. Read the string and the two integers. Call `s.substr(start, count)` and print its returned string. The constraints guarantee that the start is valid and that the requested piece fits.

Complete solution – C++23

```
#include <iostream>
#include <string>

int main() {
    std::string s;
    int start = 0;
    int count = 0;
    std::cin >> s >> start >> count;
    std::cout << s.substr(start, count) << '\n';
    return 0;
}
```

Sample input

training 2 4

Sample output

aini

Trace the sample. In training, starting at index 2 and taking 4 characters selects indices 2 through 5: aini. The number 4 is not the end index.

Why it works. The requested output is exactly the sequence specified by the position-and-count substring operation. No additional transformation is needed.

Tests and common mistakes. `count = 0` gives an empty output line. `start = s.size()` is allowed only with count zero under these constraints. Taking the whole string should preserve it.

Solution 18 – Length of the common prefix

Plan. Read both strings with two ordinary `getline` calls; there was no earlier numeric input to discard. Start the answer at zero and advance it only while the two current characters exist and match.

Complete solution – C++23

```
#include <iostream>
#include <string>

int main() {
    std::string a;
    std::string b;
    std::getline(std::cin, a);
    std::getline(std::cin, b);
    int n = static_cast<int>(a.size());
    int m = static_cast<int>(b.size());
    int answer = 0;
    while (answer < n && answer < m && a[answer] == b[answer]) {
        ++answer;
    }
    std::cout << answer << '\n';
    return 0;
}
```

Sample input

train

training

Sample output5

Trace the sample. `train` and `training` share all five characters of the shorter string. When its end is reached, the answer is 5.

Why it works. Before every increment, all preceding characters match. The loop stops at the first position that cannot extend a common prefix, so the current length is maximal.

Tests and common mistakes. Test either line empty, both lines empty, identical strings, and a first-character mismatch. A match that occurs only later in the words is not part of their common prefix.

Solution 19 – Independent maximums for several cases

Plan. Use one reusable array but a new active length and maximum for each case. Reading the case overwrites every cell that the current maximum scan will visit. Initialize the candidate from `a[0]` inside the case loop.

Complete solution – C++23

```
#include <iostream>

int main() {
    const int CAPACITY = 1000;
    int a[CAPACITY]{};
    int t = 0;
    std::cin >> t;
    for (int test = 0; test < t; ++test) {
        int n = 0;
        std::cin >> n;
        for (int i = 0; i < n; ++i) { std::cin >> a[i]; }
        int maximum = a[0];
        for (int i = 1; i < n; ++i) {
            if (a[i] > maximum) { maximum = a[i]; }
        }
        std::cout << maximum << '\n';
    }
    return 0;
}
```

Sample input

```
3
3
7 9 2
1
-5
2
-4 -8
```

Sample output

```
9
-5
-4
```

Trace the sample. The sample maximums are 9, -5, and -4. In particular, 9 from the first case must not survive into the all-negative second case.

Why it works. Within each case, the scan keeps the greatest value seen in that case. Resetting the candidate and respecting the new length makes the cases independent.

Tests and common mistakes. Test a large positive case followed by a shorter negative case. This exposes both a stale maximum and accidentally traversing the old length.

Solution 20 – Check a simple text rule

Plan. Scan the token once and set a flag when each required category is encountered. The flags only move from false to true. After the scan, combine the three flags and the minimum-length check with `&&`.

Complete solution – C++23

```
#include <iostream>
#include <string>

int main() {
    std::string s;
    std::cin >> s;
    int n = static_cast<int>(s.size());
    bool upper = false;
    bool lower = false;
    bool digit = false;
    for (int i = 0; i < n; ++i) {
        char c = s[i];
        if ('A' <= c && c <= 'Z') { upper = true; }
        if ('a' <= c && c <= 'z') { lower = true; }
        if ('0' <= c && c <= '9') { digit = true; }
    }
    bool valid = n >= 8 && upper && lower && digit;
    std::cout << (valid ? "YES" : "NO") << '\n';
    return 0;
}
```

Sample input

HnuCode23

Sample output

YES

Trace the sample. HnuCode23 has 9 characters, uppercase letters, lowercase letters, and digits, so it is accepted. A long all-lowercase token still fails.

Why it works. Each flag records exactly whether its category appeared anywhere in the input. The final conjunction is true exactly when all stated requirements are met.

Tests and common mistakes. Test exactly 8 characters, fewer than 8 with all categories, and one missing category at a time. Do not use `||` when all requirements are mandatory.

2.29 Quick reference, glossary, and revision checklist

Use these pages after studying the explanations. A short pattern is useful only when you can explain its bounds, its starting state, and what each variable remembers.

Array patterns at a glance

Assume **a** has a positive fixed capacity and its current input length is **n**. Every fragment below belongs inside **main**. The statement must guarantee that the input fits.

```
// Visit every active element.
for (int i = 0; i < n; ++i) {
    // Use a[i].
}

// Visit each adjacent pair exactly once.
for (int i = 1; i < n; ++i) {
    // Compare a[i - 1] with a[i].
}

// Visit each mirrored pair exactly once.
for (int i = 0; i < n / 2; ++i) {
    int j = n - 1 - i;
    // Compare or swap a[i] and a[j].
}
```

Task	Starting state and rule
Sum	Start at 0; add each value using a sufficiently wide type.
Count	Start at 0; increment only when the condition is true.
Minimum / maximum	For nonempty input, start from a[0] ; scan the rest.
First occurrence	Start position at -1 ; set it on a match and stop.
Last occurrence	Start position at -1 ; replace it on every match.
All elements satisfy a rule	Start a flag at true; make it false when a violation is found.
Any element satisfies a rule	Start a flag at false; make it true when a witness is found.

Two boundaries, two questions

The capacity tells you which cells exist. The active length tells you which cells belong to this input. A cell can be inside the allocation and still be the wrong cell for the current problem.

String operations at a glance

Include `<string>`. These expressions use `std::string s`; the conditions in the right column matter. Detailed rules and citations are in Sections 2.12–2.16.

Expression	Meaning / condition
<code>s.size() / s.length()</code>	Number of stored <code>char</code> elements; unsigned result.
<code>s.empty()</code>	True when the length is zero.
<code>s[i]</code>	Access a character: use $0 \leq i < \text{length}$.
<code>s.at(i)</code>	Checked access; throws for an index outside the characters.
<code>s.front(), s.back()</code>	First / last character; require a nonempty string.
<code>s += 'x'</code>	Append one character.
<code>s += "abc"</code>	Append text.
<code>s.push_back('x')</code>	Append one character.
<code>s.pop_back()</code>	Remove the last character; require nonempty text.
<code>s.clear()</code>	Make the string empty.
<code>s.substr(pos, count)</code>	Copy up to <code>count</code> characters from <code>pos</code> ; <code>pos</code> may equal the length but must not exceed it.
<code>s.find("ab")</code>	First match's index, or <code>std::string::npos</code> .
<code>a == b, a < b</code>	Compare string contents; ordering is lexicographical, not numeric.

Which input method belongs here?

One token with no spaces: use `std::cin >> s`; **A whole line:** use `std::getline(std::cin, s)`; **A number on its own line, then a text line:** read the number, discard the remainder of that line once, then call `getline`. Do not discard a line repeatedly when you are already using `getline`.

Safe bounded traversal

In this chapter, every processed string length is at most 100000, which fits the `int` used by the training compiler.

```
int n = static_cast<int>(s.size());
for (int i = 0; i < n; ++i) {
    char current = s[i];
    // Examine current without changing s's length.
}
```

Do not keep using an old `n` after an operation changes the string's length. Do not write at `s[s.size()]` to append a character.

A small glossary

Term	Meaning in this chapter
Element	One stored value, such as <code>a[2]</code> or <code>s[2]</code> .
Index	The number used to identify an element; normally starts at 0.
Active length	Number of elements belonging to the current input.
Capacity	Number of available cells in a fixed array; a string manages its own storage capacity.
Traversal	Visiting elements one at a time.
Linear search	Looking through the elements in sequence for a target.
Flag	A boolean variable remembering whether a condition has occurred or still holds.
Sentinel result	A special non-answer, such as <code>-1</code> for a missing index.
In-place change	A change made to the existing stored sequence.
Matrix	A table described by row and column indices.
Transpose	Exchanging a table's rows and columns.
Token	A unit of input separated by whitespace when using <code>>></code> .
Concatenation	Joining strings end to end.
Lexicographical order	Compare from the beginning; the first differing character decides, or the shorter matching prefix comes first.
Palindrome	A sequence that is unchanged when read in reverse.
Substring	A consecutive part of a string.
Prefix / suffix	A consecutive part at the beginning / end.
Subsequence	A sequence formed by keeping some elements in their original order, with gaps allowed.
Half-open range	A range including its start and excluding its end, such as $[0, n)$.

Use precise words when explaining a solution

“The value at index 3” is clearer than “the third number” when zero-based indexing matters. “Count matching elements” is different from “sum matching elements.” A precise sentence often prevents an incorrect loop.

Can you do these without looking at a solution?

Mark a line only after writing or explaining a small example yourself.

1. Distinguish an element's value from its index, and translate a one-based position.
2. Declare a fixed-capacity array, initialize it, and explain why a runtime-sized raw array is not standard C++23.
3. Read and print exactly the active elements, including the empty case when allowed.
4. Choose a type that can hold the largest possible sum, not just one input value.
5. Find the minimum, maximum, and a target; explain first-versus-last tie handling.
6. Copy, compare, replace, or reverse array values without losing needed information.
7. Trace adjacent and mirrored indices for one-element, odd-length, and even-length input.
8. Read a rectangular matrix and compute row, column, and diagonal totals.
9. Explain the difference between `'7'`, `"7"`, and `7`.
10. Read a token or a full line and handle the transition from numeric input deliberately.
11. Count or filter characters, convert a digit character, and keep leading zeros when appropriate.
12. Compare strings, take a substring, and handle an unsuccessful `find`.
13. Check a palindrome and count words using the statement's exact definition.
14. Reset state for several test cases and test cases of different lengths.

A final independent task

Read a nonempty line of at most 100000 lowercase letters and spaces. Ignore the spaces, and decide whether the remaining letters form a palindrome. A line of only spaces should produce **YES**. Before coding, explain which two patterns you will combine and why the filtered text may be empty.

Self-check. `"n u r s e s r u n"` should produce **YES**; `"hello world"` should produce **NO**. Build the filtered string first, then compare its mirrored pairs. This is a suggested independent exercise, not one of the 20 fully solved practice problems.

Ready for Session 3

The next step is to package useful work into functions and describe how the amount of work grows as the input grows. For now, focus on explaining one correct scan, its variables, and its boundaries. You do not need to finish every optional extension before moving on.

2.30 Technical notes and references

These notes explain the conventions behind the beginner-facing examples. They are a reference for a second reading, not additional material to memorize before attempting the practice.

Language version and scope

The examples target **C++23**. Standard-language references below point to the fixed **N4950 C++23 working draft**, rather than a moving draft that may include later changes. The linked draft is a public HTML rendering of committee wording; it is not a tutorial. GNU and Microsoft references are their official implementation documentation.

The chapter teaches built-in arrays and `std::string` first. Sorting, `std::array`, range-based loops, and additional string operations are clearly marked as optional. It does not require user-defined functions, vector, maps, sets, prefix sums, or frequency arrays. A library operation is introduced where used, without requiring the later lesson on writing your own functions.

Integers, lengths, and input guarantees

The supplied programs were compiled on a GNU environment with a 32-bit `int` and a 64-bit `long long`. C++ does not require `int` to be exactly 32 bits. This chapter's 10^9 integer inputs and 100000-character bounds assume an `int` wide enough for them, as on the tested training environment. Totals and products that can exceed `int` use `long long`.

Unless stated otherwise, input follows the format and constraints in the task. The examples are contest solutions, not interactive input-validation applications. In particular, a fixed array is only sufficient when the promised length fits its capacity.

Converting a string length to `int` is justified here by the stated small bound. It is not a general recommendation to narrow an arbitrarily large container size. When reasoning about a backward loop, remember that an unsigned size does not become negative when decremented below zero.

Text means simple characters in these problems

The tasks use the stated ASCII letters, digits, and spaces. A `std::string` contains `char` elements; its length is not generally the number of displayed letters in an arbitrary encoded language. Byte-by-byte reversal and character classification are therefore not presented as general Unicode text-processing techniques. Decimal digit conversion relies on the consecutive digit values required by C++; the letter-range tests use the explicit ASCII assumption.^{[2-8][2-9]}

The end of a string is not an extra editable character

For character traversal, use indices strictly smaller than the string length. C++23 has a special rule permitting `s[s.size()]` to refer to a null character; changing that character to a nonzero value is undefined behavior. It is not an append operation. `s.at(s.size())` throws instead. `front`, `back`, and `pop_back` require a nonempty string.^{[2-11][2-12]}

What was checked for this edition

The source package includes **38 complete Session 2 examples** and **20 complete practice solutions**. All 58 compiled successfully with GCC 14.2.0 in C++23 mode, with strict diagnostics. All passed **5552 input/output checks**: 134 fixed sample/boundary checks and 5418 generated or exhaustive-small-case checks.

Tests include missing targets, duplicate values, all-negative arrays, empty inputs where permitted, one-element inputs, odd/even lengths, maximum capacities, wide sums, rectangular matrices, blank lines, leading/trailing spaces, and repeated test cases. Expected results for generated inputs are produced independently by the Python test runner.

The verification build used these flags:

```
-std=c++23 -pedantic-errors -Wall -Wextra  
-Wconversion -Wshadow -Werror -O1 -g  
-D_GLIBCXX_ASSERTIONS -fsanitize=undefined  
-fno-sanitize-recover=all
```

The normal learning build does not require all of these flags. They record the checking configuration for the distributed code. Undefined-behavior checks and library assertions can reveal faults on executed paths; passing tests is not a mathematical proof for every allowed input. Deliberately wrong fragments are labeled and are not compiled as working solutions.^{[2-27][2-28]}

Reproduce the checks or rebuild the book

The source package includes `tests/verify_session_02.py`, the fixed test manifest, and a JSON verification report. Run the script from the project root using Python 3 and a C++23-capable GNU compiler. The README gives the exact commands.

Compile `session_02.tex` for the separate Session 2 PDF. Compile `book.tex` for the collected edition containing Sessions 1 and 2. Both include this same chapter and the same external C++ files, so code changes stay synchronized. The original Session 1 chapter remains part of the project.

Reading the reference markers

A superscript such as ^[2-11] links to the corresponding entry below. The first number identifies Session 2, so references do not collide with Session 1 when the chapters are collected. References support the language and library rules; the teaching examples, exercises, and their explanations are written for this training.

Language and standard-library sources

[2-1] C++23 N4950: built-in array declarators

<https://timsong-cpp.github.io/cppwp/n4950/dcl.array>

[2-2] C++23 N4950: aggregate initialization

<https://timsong-cpp.github.io/cppwp/n4950/dcl.init.aggr>

[2-3] C++23 N4950: initialization and indeterminate values

<https://timsong-cpp.github.io/cppwp/n4950/dcl.init.general>

[2-4] C++23 N4950: subscripting

<https://timsong-cpp.github.io/cppwp/n4950/expr.sub>

[2-5] GCC: variable-length arrays as an extension

<https://gcc.gnu.org/onlinedocs/gcc/Variable-Length.html>

[2-6] C++23 N4950: array-to-pointer conversion

<https://timsong-cpp.github.io/cppwp/n4950/conv.array>

[2-7] C++23 N4950: assignment operators

<https://timsong-cpp.github.io/cppwp/n4950/expr.ass>

[2-8] C++23 N4950: character sets and digit values

<https://timsong-cpp.github.io/cppwp/n4950/lex.charset>

[2-9] C++23 N4950: the basic string class

<https://timsong-cpp.github.io/cppwp/n4950/basic.string>

[2-10] C++23 N4950: string size, length, and capacity

<https://timsong-cpp.github.io/cppwp/n4950/string.capacity>

[2-11] C++23 N4950: string element access

<https://timsong-cpp.github.io/cppwp/n4950/string.access>

[2-12] C++23 N4950: string modifiers

<https://timsong-cpp.github.io/cppwp/n4950/string.modifiers>

[2-13] C++23 N4950: string extraction and getline

<https://timsong-cpp.github.io/cppwp/n4950/string.io>

[2-14] C++23 N4950: unformatted input and ignore

<https://timsong-cpp.github.io/cppwp/n4950/istream.unformatted>

[2-15] C++23 N4950: the ws manipulator

<https://timsong-cpp.github.io/cppwp/n4950/istream.manip>

[2-16] C++23 N4950: string concatenation

<https://timsong-cpp.github.io/cppwp/n4950/string.op.plus>

[2-17] C++23 N4950: string comparison

<https://timsong-cpp.github.io/cppwp/n4950/string.compare>

[2-18] C++23 N4950: substr

<https://timsong-cpp.github.io/cppwp/n4950/string.substr>

[2-19] C++23 N4950: string construction and substring bounds

<https://timsong-cpp.github.io/cppwp/n4950/string.cons>

[2-20] C++23 N4950: string searching

<https://timsong-cpp.github.io/cppwp/n4950/string.find>

[2-21] C++23 N4950: sort

<https://timsong-cpp.github.io/cppwp/n4950/sort>

[2-22] C++23 N4950: reverse

<https://timsong-cpp.github.io/cppwp/n4950/alg.reverse>

[2-23] C++23 N4950: range-based for statements

<https://timsong-cpp.github.io/cppwp/n4950/stmt.ranged>

[2-24] C++23 N4950: the array container

<https://timsong-cpp.github.io/cppwp/n4950/array>

[2-25] C++23 N4950: character handling declarations

<https://timsong-cpp.github.io/cppwp/n4950/cctype.syn>

[2-26] GCC: language-standard options

<https://gcc.gnu.org/onlinedocs/gcc/C-Dialect-Options.html>

[2-27] GCC: warnings and strict conformance diagnostics

<https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html>

[2-28] GCC: instrumentation and sanitizer options

<https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html>

[2-29] Microsoft CRT: tolower and its input restrictions

<https://learn.microsoft.com/en-us/cpp/c-runtime-library/reference/tolower-tolower-towlower-tolower-l-towlower-l>

[2-30] C++23 N4950: prefix, suffix, and containment operations

<https://timsong-cpp.github.io/cppwp/n4950/string.view.ops>

[2-31] C++23 N4950: character-array initialization

<https://timsong-cpp.github.io/cppwp/n4950/dcl.init.string>

Additional detail for [2-20]. The searching rules delegate to string-view searching, including the first matching index and `npos` when absent: <https://timsong-cpp.github.io/cppwp/n4950/string.view.find>.